Erbium : Reconciling languages, runtimes, compilation and optimizations for streaming applications

Cupertino Miranda

UNIVERSITÉ PARIS SUD
Comprendre le monde, construire l'avenir®

UNIVERSITÉ DE PARIS-SUD
ECOLE DOCTORALE INFORMATIQUE

In Partial Fulfillment of the Requirements for the Degree of
DOCTOR OF PHILOSOPHY
Discipline: Computing science

defended on 11/02/2013

Cupertino MIRANDA

ERBIUM: RECONCILING LANGUAGES, RUNTIMES, COMPILATION AND OPTIMIZATIONS FOR STREAMING APPLICATIONS

Thesis supervisor: Dr. Albert Cohen

Thesis committee:

M. Albert Cohen	Senior Research Scientist at INRIA
M. Alain Darte	Senior Research Scientist at CNRS
M. Marc Duranton	Senior Research Scientist at CEA
M. Daniel Etiemble	Professor at University Paris Sud
M. Luciano Lavagno	Professor at Politecnico di Torino

To honor the memory of my grandfather Cupertino Duarte Miranda

Acknowledgments

This thesis would not be possible without the support of a handful of people both at a technical and/or emotional level.

I would like to thank Albert Cohen for believing in me, taking me as his PhD student, guiding me through research but also allowing me to follow my own research path and interests. Without his guidance, knowledge and feedback this PhD would not have been achievable.

I would like to express my greatest appreciation to Alain Darte and Luciano Lavagno for their detailed manuscript revisions and their extremely valuable comments and suggestions. Those were really important and significantly improve the quality of this document.

I would like to show my appreciation to Daniel Etiemble for leading my thesis defense committee, but also to all the committee members for the great questions and remarks.

I thanks Marc Duranton, not only for being part of my thesis committee, but also for his contribution to the presented thesis topics and very productive brainstorming sessions.

For everyone from IBM Haifa, my big appreciation for being so friendly and for all the excellent times. In particular to Uzi Shvadron for being a mentor/friend and for all the great technical interactions. For both Uzi and Ayal Zaks, thank you for the great guided weekend visits through the country.

A big thank you to my parents for always supporting my decisions, but more importantly, for always motivating my curiosity throughout my childhood. They allowed me to learn through experimentation, even when sometimes finishing with soldering iron burns and some pain.

Last but definitely not least, I would like to thanks Paula for being part of my life (my other-half), supporting me in any good or bad decisions, but also providing me the confidence and the perseverance to finalize the thesis work by giving me the strength to persist, sometimes shaking me up when I had questioned myself.

This thesis was financially supported by Fundação para a Ciência e a Tecnologia (FCT) PhD grant with the reference SFRH / BD / 37332 / 2007.

Abstract

As transistors size and power limitations stroke computer industry, hardware parallelism arose as the solution, bringing old "forgotten" problems back into equation to solve the existing limitations of current parallel technologies. Compilers regain focus by being the most relevant "puzzle piece" in the quest for the expected computer performance improvements predicted by Moore's law — no longer possible without parallelism. Parallel research is mainly focused in either the language or architectural aspects, not really giving the needed attention to compiler problems, being the reason for the weak compiler support by many parallel languages or architectures, not allowing to exploit performance to the best.

This thesis addresses these problems by presenting: Erbium, a low level streaming data-flow language supporting multiple producer and consumer task communication; a very efficient runtime implementation for x86 architectures also addressing other types of architectures; a compiler integration of the language as an intermediate representation in GCC; a study of the language primitives dependencies, allowing compilers to further optimise the Erbium code not only through specific parallel optimisations but also through traditional compiler optimisations, such as partial redundancy elimination and dead code elimination.

Résumé

Frappée par les rendements décroissants de la performance séquentielle et les limitations thermiques, l'industrie des microprocesseurs s'est tournée résolument vers les multiprocesseurs sur puce. Ce mouvement a ramené des problèmes anciens et difficiles sous les feu de l'actualité du développement logiciel. Les compilateurs sont l'une des pièces maitresses du puzzle permettant de poursuivre la traduction de la loi de Moore en gains de performances effectifs, gains inaccessibles sans exploiter le parallélisme de threads. Pourtant, la recherche sur les systèmes parallèles s'est concentrée sur les aspects langage et architecture, et le potentiel reste énorme en termes de compilation de programmes parallèles, d'optimisation et d'adaptation de programmes parallèles pour exploiter efficacement le matériel.

Cette thèse relève ces défis en présentant Erbium, un langage de bas niveau fondé sur le traitement de flots de données, et mettant en œuvre des communications multi-producteur multi-consommateur; un exécutif parallèle très efficace pour les architectures x86 et des variantes pour d'autres types d'architectures; un schéma d'intégration du langage dans un compilateur illustré en tant que représentation intermédiaire dans GCC; une étude des primitives du langage et de leurs dépendances permettant aux compilateurs d'optimiser des programmes Erbium à l'aide de transformations spécifiques aux programmes parallèles, et également à travers des formes généralisées d'optimisations classiques, telles que l'élimination de redondances partielles et l'élimination de code mort.

Contents

Chapter 1

Introduction - Problem Statement

During many years, the semi-conductor industry took advantage of the exponential increase in transistor density and of the decrease of their cost, latency and power consumption, to add functionalities and performance-enhancing features to the newer generation of processors. These advances, together with further increases in clock speeds, provided the industry with enough technological innovation to justify consumers recycling old machine and buy faster new ones.

Soon enough technological progress started to reach physical limitations, where feature improvements and size reduction no longer translated into higher clock-speeds and where power consumption hits thermal dissipation and battery life limits. As a reaction, the industry focused its attention on parallel chip-multiprocessor architectures, also known as multicore processors, in an attempt to take advantage of Moore's law and get performance at lower clock speeds and in a more power efficient manner. Moreover, in order to package the multiple processors in single chip, it was necessary to simplify the individual core complexity, allowing the full processors to maintain its price and total number of transistors. This new design decision puts an end to the legacy application yearly free performance improvements, forcing application engineers to redesign (parallelization, locality optimization) older codes, exploiting the computational power of the new generation of parallel processor architectures.

1.1 Computer system abstractions

Computer systems can be understood as a combination of abstraction layers composing what we know as a computer system. Each abstraction lowers the complexity involved in the system programmability, being also responsible for the differentiation of what is known as software and hardware.

Computer architectures, often seen as the most fundamental computer component, is also an abstraction, hiding its logic circuitry through its instruction set architecture. The assembly language is its immediate connecting abstraction, mapping the sequence of operations into sequences of bits representing the executing instructions.

Throughout the years, computers were further abstracted with the introduction of operating systems and higher level programming languages.

1.1.1 Architectures

Long before plain consumers first heard of parallelism they were already executing there own sequential applications in parallel. Processors circuitry by design parallelizes operations at its operands bit-level. Bit-level parallelism is the effect of such low-level circuit design parallelization.

Apart from the bit-level parallelism, sequential processors also implement instruction level parallelism (ILP). ILP comes from the partitioning of an instruction execution into several independent pipeline stages. Examples of such processors are the IBM 7094 with only 2 stages and the more recent and complex Pentium 4 with 32 stages, yet still both executing a single thread at a time.

Other type of sequential executing architectures, also with parallelism, are the superscalar processors, having both a long instruction pipeline and multiple functional units. Multiple instructions are decoded simultaneously and depending on availability of functional units, the instructions are executed simultaneously. One of the main difficulties with superscalar machines is within the design of the instruction dispatcher, which must validate for instruction dependencies and resources availability.

The limitations on superscalar architectures gave birth to Very Long Instruction Word (VLIW) architectures. VLIW enforces static computed ILP during compilation, simplifying the architecture and its penalty for hardware instruction scheduling.

The more recent generation of architectures, such as latest Intel architectures are composed of multiple similar cores sharing memory through its cache hierarchy of memories. IBM's Cell Broadband Engine [42] has a more heterogeneous approach being composed of a PowerPC microarchitecture connected with 8 simplified accelerator processors, each with its independent memory space.

Memory

One of the key components of any architecture is its memory model and its implementation. Depending on the envisioned architecture properties, either or both of the memory models are used:

- **Shared memory** model refers to the connection of independent processors to the same memory space, sharing data. This memory model typically involves higher latency and bandwidth between the processors and memory. The processors implement caches to hide its latency and bandwidth limitations.

- **Distributed memory** refers to the systems with more then one memory space. A single memory space can be associated with one or more processors or cores and only these can access it. In a distributed memory system, the application should explicitly perform any needed memory transfer.

Later generation of architectures, more precisely heterogeneous architectures, have a mixed implementation of both models and are typically composed of one or more general purposes Core CPUs, using traditional shared memory and several accelerators each containing an independent and private memory space (distributed memories). An example of such an architecture is the previously mentioned IBM Cell BE processor.

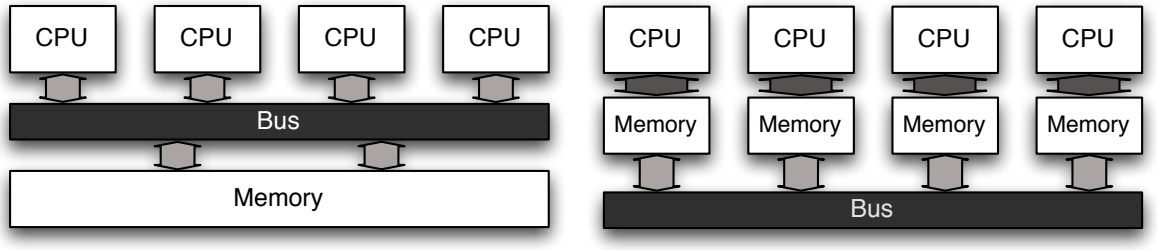


Figure 1.1: Memory model abstract diagrams: Shared memory (left) vs. Distributed memory (right).

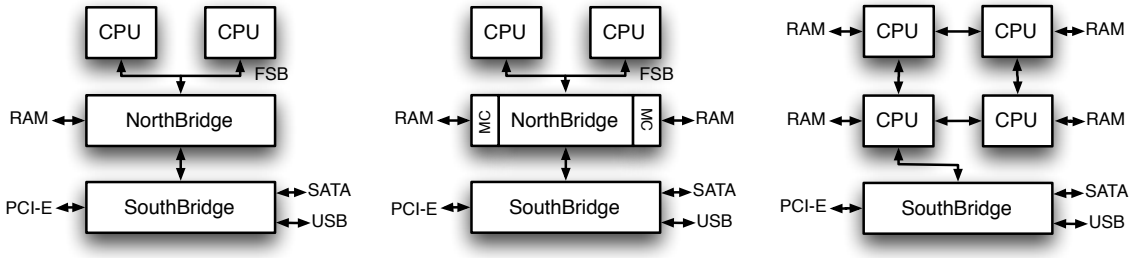


Figure 1.2: Examples of memory to CPU connectivities.

Memory connectivity

General purpose computers have more then one controller chip integrating and connecting facilities both for CPUs and memory, namely NorthBridge and SouthBridge, controlling other external protocols, as Figure 1.2 (left) exemplifies. In this most simple example, CPUs connect via a bus to NorthBridge chip. The NorthBridge contains a memory controller, logically connecting with the available RAM modules. Different types of RAM need different type of memory controllers. Memory controllers have a great impact in memory latency times and bandwidth.

Apart from communicating with RAM, North-Bridge also connects to a South-Bridge chip, which provides connectivity to other types of buses and eventually other devices. All the data communicating between CPUs as well as any RAM access from South-Bridge devices must travel through North-Bridge.

In Figure 1.2 (middle) design, the NorthBridge no longer contains integrated memory controllers, but instead connects to independent parallel memory controllers with independent memory buses and therefore greater communication bandwidth.

The design in Figure 1.2 (right) also increases bandwidth, but this time NorthBridge was removed from the system and each CPU includes its own memory controller connecting directly their own respective module. With this technology, memory accesses avoid the cost of communicating through NorthBridge and execute much faster. Although, as memory in this systems is not uniform (at the CPU context), whenever a CPU needs some memory region controlled by other CPU, performance is degraded by each of the memory controllers the data has to travel through. CPU performance is not impacted by memory accesses between CPUs as it is asynchronous from the code execution.

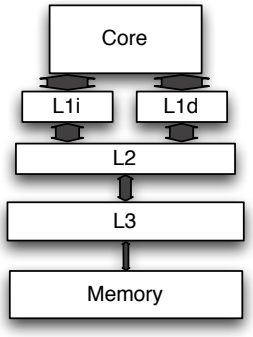


Figure 1.3: Cache hierarchy in a single processor CPU.

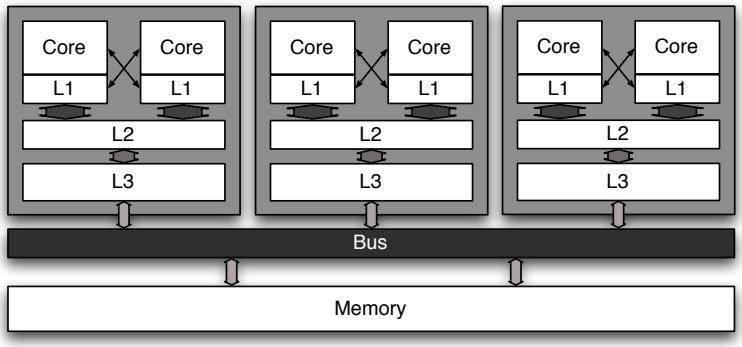


Figure 1.4: Cache diagram in a multiprocessor shared memory system

Caches, Coherency

No matter how close memory modules or controllers are from the actual CPU, load and store instructions (memory accesses) take many more clock cycles than simple register operating instructions. Moreover, there are a broad variety of memory types, ranging from very cheap and slow (typical PC memory RAM modules), to expensive but fast internal to CPU circuitry, representing the processor registers. Therefore most traditional general purpose architectures contain several layers of memories, which minimize access times for most recent accessed memory addresses, namely caches. Caches are smaller, faster and closer to CPU memories used to hide the latency and bandwidth memory accesses overheads, by replicating latest used memory addresses data. Caches are organized into hierarchies of memories, each having different sizes and speeds. The closer those memories are from the CPU, the faster and smaller they get. Figure 1.3 presents an overview of the cache hierarchy of a most common single core general purpose processor. Figure 1.4 is a similar overview of the cache hierarchy however in a shared memory multi-core and multi-processor system.

When a CPU decodes an instruction that needs to perform a memory load, it checks the content of the cache for a possible availability (cache-hit) within its closer memory (L1 cache). If the needed memory region is not available (cache-miss) it verifies in the higher level cache memories and eventually copy the address from main memory into cache.

The caches are partitioned in blocks of memory (cache-lines) where a specific memory region is always associated to the same cache-line. The selection of cache-line is computed based on a subset of the memory address. When a used cache-line requires to store a different memory-region, the current content of the cache-line is copied (evicted) to an higher-level cache memory, which eventually, when cache storage capacity per cache-line is fully taken, is copied back to memory. To reduce the number of evictions by conflicting cache-line memory addresses, caches are defined as set-associative. This allows the same cache-line address to store more then one simultaneous cache-line and still hide memory latency when accessing disjoint memory regions that would compete for the same cache-line. When the CPU needs to write some memory region, by default it will only change the content of the respective cache-line as well as mark the content of this cache-line as dirty. Marking a cache-line as dirty allows to reduce the number of copies back to memory by identifying which cache-lines were ever modified.

Single processor architectures clearly benefit with caches, considering that both latency and bandwidth are improved significantly by moving data closer to the CPU, reducing the expensive RAM communications. On the other hand, in multi-core architectures it is not as easy to understand how caches can be managed coherently and with minimal overhead. As it is too expensive to share a cache between independent cores (at least at its lowest level), every CPU core has a private L1 cache. Such cache privatization introduces coherency problems. MESI (Modified, Exclusive, Shared, Invalid) coherency protocol approaches this problem by marking each cache-line with one of the 4 MESI states. Depending on the actual cache-line and on pending operations for the cache-line, the MESI protocol enforces an action on the actual cache-line data and eventual state change to all of the core private caches. Although the coherency algorithms execute asynchronously to the processor execution, through its independent hardware logic, multiple concurrent accesses to same memory addresses result in significant processor slowdowns resulting from the data unavailability.

Consistency

As architectures were accelerated through the increasing clock speeds and memory were still bounded to same access times, caches improved performance through parallelization, partitioning its memory into independent and simultaneously accessible banks. The partitioning improves cache access times, but unfortunately also produces unpredictability in memory operation ordering in the parallelized cache banks. This unpredictability contributes to definition of the architecture consistency model.

Sequential consistency is the most restrictive form of memory consistency enforcing that all memory operations are visible by all the CPU's in a single global ordering. Unfortunately sequential consistency is not very effective due to its global order enforcement, limiting the system ability to exploit parallel memory operations. Instead, architectures have evolved into more flexible memory consistency models. Such models are considered weak since sequential operation ordering is not guaranteed in at least one of the consistency rules. Table 1.1 lists these rules and provides an overview of the consistency rules for several well known architectures.

Weakly consistent architectures enforce sequential consistency through the usage of memory barriers primitive instructions. Memory barriers wait for the completion of a specific type of memory operation before further thread execution.

As visible through Table 1.1, there are many distinct memory consistency models. However, as mentioned by McKenney [53], there are ground rules that all the weak consistent architectures respect:

- Each CPU always perceives its own memory operations as occurring as in the program code ordering.
- Store operations are only reordered with load operations if those only access a different location than the stored one.
- Simple aligned loads and stores are atomic. Small size loads or stores are not interruptible and always execute completelly. As an example, 64 bit aligned loads or stores are atomic in 64 bit architectures but not in 32 bit ones.

Memory consistency is a very important property of the architecture, considering processor concurrency and synchronization algorithms. Depending on the architecture consistency

Type	Alpha	ARMv7	PA-RISC	POWER	SPARC RMO	SPARC PSO	SPARC TSO	x86	x86 oostore	AMD64	IA64	zSeries
Loads reordered after Loads	✓	✓	✓	✓	✓				✓		✓	
Loads reordered after Stores	✓	✓	✓	✓	✓				✓		✓	
Stores reordered after Stores	✓	✓	✓	✓	✓	✓			✓		✓	
Stores reordered after Loads	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Atomic reordered with Loads	✓	✓		✓	✓						✓	
Atomic reordered with Stores	✓	✓		✓	✓	✓					✓	
Dependent Loads reordered	✓											
Incoherent Instruction cache pipeline	✓	✓		✓	✓	✓	✓	✓	✓		✓	✓

Table 1.1: Memory consistency weaknesses of several widely known computer architectures. Architectures marked with ✓are weaken in the respective row property. Inspired by [53].

model, synchronization algorithms can be tuned to reduce overheads and achieve better performance.

Direct Memory Access (DMA)

Distributed memory architectures might not rely on caches but rather on very fast and small local to processor memories, as is the case of IBM's Cell Processor [42] SPU elements. Its cores have no abstraction to a global memory. Instead, data transfers are explicitly requested by the running application which is responsible to guarantee data availability in all of the executing processors local memories. These processors do not have caches as their memories are smaller and embedded into the CPU. Figure 1.5 presents how the Cell processor subsystems and cores are internconnected.

Such architectural design makes its cores less convenient for general purpose usage due to the lack of code portability and extreme steep learning programming curves. As there is no abstraction to a global shared memory, there is no need for coherency checks and its hardware complexity is simplified, leading to faster data transfers and a more power efficient architecture.

1.1.2 Languages

Programming languages are abstractions to all of the difficulties involved within programming computers. During many years computer engineers argued about the interest of higher abstraction programming languages when comparing with the closer to hardware assembly languages. Today, almost no application is developed in assembly language thanks to the great improvements introduced by higher level programming languages and optimizing compilers. Unfortunately such improvements are still not reachable to parallel applications and languages.

Many programming languages, extensions and application programming interfaces (APIs) have been designed in an attempt to address parallelism difficulties and tuning nuances. Such

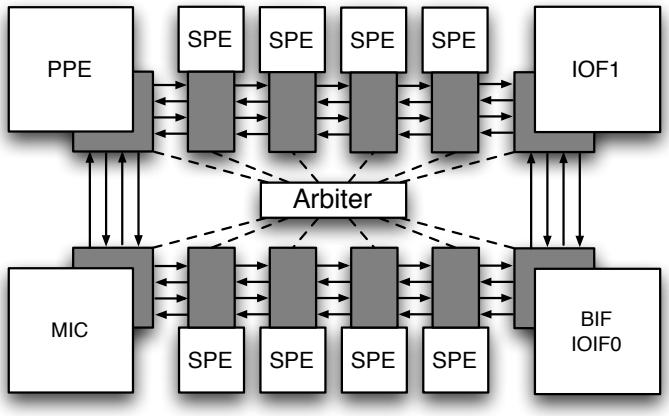


Figure 1.5: Cell interconnect diagram presenting the interconnection between all the processor subsystems and cores.

```
1 for (i = 1; i < N; i++) {
2   A = V[i] * C1;
3   B = V[i] * C2;
4   C[i] = A +B;
5 }
```

```
1 for (i = 1; i < N; i++) {
2   C[i] = A[i] * B[i];
3 }
```

```
1 for (i = 1; i < N; i++) {
2   B[i] = A[i] * C1;
3 for (i = 1; i < N; i++) {
4   C[i] = B[i] * C2;
5 }
```

Figure 1.6: Sequential code containing task (left), data (middle) and pipeline (right) parallelism

designs, similarly to what already happened within sequential languages, vary in abstraction granularity, going from hard to code but very expressive languages to the very easy although very limited ones. The more expressive and closer to hardware a language is, the greater is its learning curve and the less portable the language might be. On the other hand, simpler languages limit the programmer to specific types of applications, reducing the number of possibly supported applications. Nevertheless, such languages tend to evolve around constraints that allow the language to be statically predictable and easy to optimize.

Parallelism

Common parallel languages use higher level abstractions representing the following forms of parallelism:

- Task parallelism refers to parallelism that occurs from two totally independent branches on the original application.
- Data parallelism refers to parallelism that occurs within the same computation but where different computational data ranges have no dependencies.
- Pipeline parallelism occurs when a computation can be partitioned into several unidirectional dependent execution blocks.

Figure 1.6 presents three code examples, each containing a different type of parallelism form. The left example is a task parallel application. The statements computing A and B have no dependences and can be computed in parallel. On the other hand, the statement in Line 4 needs the result of both statements, requiring some form of synchronization in order

to guarantee the execution of previous parallel statements. Such type of parallelism is usually associated with barrier synchronization primitives.

Figure 1.6 (middle) is an example of data parallelism. Statements inside the loop are independent between loop iterations, i.e., computation of $C[i]$ can be executed for all i values concurrently. Moreover, data parallelism earns its name from the independence of its input and output data.

Figure 1.6 (right) is an example of pipeline parallelism. Pipeline parallelism is a restricted form of task parallelism, in which statements are not task parallel because the result (output) of one of the task is the input of the next. Each of the parallel tasks performs a stage of the algorithm and provides the data for the next task to proceed. An example of pipeline parallelism is the instruction level parallelism (ILP) commonly implemented in hardware.

In all forms of parallelism, concurrent tasks eventually require to synchronize in order to announce work completion. The most common three forms of synchronizations are:

- Mutual exclusion, commonly known as Mutex, allows an application to serialize access to resources.
- Event synchronization allows a thread or group of threads to wait for a concurrent thread signal.
- Barriers allow a group of threads to synchronize at specific program points. These are used in algorithms that need to proceed in phases, such as task parallelism.

Many kinds of parallel languages were created in an attempt to harness as many target architectures and parallelism forms. However, such languages tend to focus in specific fields of applicability, as is the case of streaming for video processing (stream related) applications and transactional memory for financial and database related applications. It is unlikely that any future general purpose higher level language would be able to create better optimized code than any of the field specific programming languages.

1.1.3 Operating Systems - Execution Models

Apart from the architecture instruction sets and its properties, computer systems are composed of yet another abstraction layer. Such layer is typically defined by an operating system and device drivers, abstracting applications from complexities such as process execution, multitasking, resources and devices management. More relevant to parallelism are its abstraction to threads and interprocess communication.

As referred by Tanenbaum in [79], threads are like operating system processes, although sharing the same memory and program address space. In other words, a process is composed of one or more threads, sharing the same address space, global variables, IO and more. Each thread is given by the operating system a different program counter and stack space.

A single core processor, when executing a multi-threaded application, appears to execute its threads simultaneously. It only happens because the threads are constantly in execution switching, creating the illusion of a simultaneous execution. Context switching implies the operating system to backup the registers and current program counter and restore the registers from the planned to execute thread, just before jumping into the switched thread program counter. Context switch is controlled by the operating system scheduling algorithm.

Threading systems are typically classified as:

- User Space, when implemented through a user level space. Threads of this type have significant performance benefits when comparing to alternatives. However, they are still bounded to a single hardware thread, disabling concurrent execution. Moreover, thread context switches always occur explicitly through runtime system calls, possibly leading to deadlocks if any of the threads requires to perform operating system blocking calls, such as waiting for IO.

- Kernel Space threads, as the name suggests, are controlled by the operating system kernel, allowing full use of the machine processing power by supporting executing threads concurrently. The kernel performs thread context switches automatically based on process scheduling policies.

In order to take full advantage of user level threads and still use full hardware resources, hybrid implementations using both kernel and user level threads are commonly used. Kernel level threads are created to exploit all of the concurrent processor cores. Each kernel thread executes an instance of the user level threading library. The application instantiates parallel code in the available and distributed user-level threading libraries.

The kernel level thread schedulers perform scheduling based on either blocking kernel level calls or through some time allocation algorithms, sometimes based on priority rules, in order to better load balance the threads execution. Such scheduling algorithms are optimized to execute blocking threads, such as, threads waiting for resources such as input operations (keyboard events, etc.) or exceptions. However, if a thread requires to wait based on a shared memory variable event from a counter part thread, none of the available blocking system calls is sufficiently fast when comparing to actual busy waiting (thread spinning). Moreover, in busy wait scenarios and when the number of threads surpasses the number of available hardware threads (CPU cores), process schedulers too oftenly wake threads not ready for execution, producing expensive context switch overheads without the application progress.

User level threads provide a better control of thread scheduling, i.e., as its implementation is at user level space, precise runtime implementation can redefine the scheduling policies based on the known constructs of the runtime environment. However, an user level threading system implies complexities such as its explicit context switch or its inability to perform context switches when calling operating system blocking functions.

1.2 Streaming

As mentioned through the chapter, current generation of architectures are increasingly becoming less complex and more parallel. Its simplification translates in an increasing necessary effort from the language and compiler developers, to create newly optimizations and abstractions to achieve performance through parallelism, exploiting these new generation of architectures.

The streaming programming concept not only surpasses the boundaries of any particular language but also crosses the boundaries of distinct abstract models of computation, as described by Stephens [77] in its study on the rule of streaming in programing languages. The streaming languages/applications are defined as graphs, where nodes are the concurrent

tasks and edges are the FIFO channels connecting different tasks. The different streaming models differ in several key properties impacting both execution model, channels and the static predictability of the model.

The most known models of computation are Kahn Process Networks (KPN) and Static Data-flow (SDF). A Kahn process network (KPN) [44] is defined based on the composition of several communication tasks. Tasks communicate through unbounded FIFO channels, where only read operations of the channels are blocking, forcing tasks to wait when no data is available, much like a *pop* queue operation. Write operations do not block considering its unbounded FIFO channels. As tasks have no way to peek channels and possibly change its control-flow based on the upcoming data, KPN tasks have the properties of *pure* functions, where its output is only a consequence of its input values or in other words deterministic. Moreover, as tasks are deterministic so is a network of tasks. However, as the rate of communication is defined by the amount of *push* or *pop* operations, the model is vulnerable to deadlocks, in case of cyclic networks.

In SDF, tasks are defined with static consumption and production rates. Moreover, task execution can be defined based on static scheduling deduced by its task graph and static defined rates. SDF is well known for its static verifiability and optimize-ability [50]. Buffer allocation and process scheduling can be determined at compilation time between other optimizations, such as process fusion, allowing a better and greater control on application load balancing. Cyclo-static data-flow (CSDF) [16] extends SDF by supporting tasks with cyclic static rates.

Streaming applications provide programmers with sufficient tools to express all presented types of parallelism, i.e., task, data and pipeline parallelism. Furthermore, streaming languages through its abstractions allow to identify the parallel computation regions and its dependencies, typically expressed as a graph. SDF or CSDF computational model languages are statically predictable allowing the compiler to schedule concurrent tasks by fusing (combining multiple dependent pipeline tasks) or task blocking (coarsening a task data communication granularity). Static scheduling results in an adjustment of the initial partitioning of the application to the actual target architecture resources, threading system and schedulers.

Moreover, streaming also exposes channel properties such as the size of the communicated data and the rate of communication, exposing also the necessary information to further optimize memory to the slowest type of architecture instructions, i.e., memory loads and stores. This allows to predict, identify, and adjust the rate of communication and channels (buffers) size, adjusting applications to the architecture cache levels, cache-line and memory sizes. Its predictability provides languages with the means to improve data locality, anticipating memory copies through cache pre-fetching, hiding memory latency and reducing the cache misses when accessing the channels data. These cache based optimizations are also available and required in distributed memory architectures where data transfers are explicit, forcefully enforcing the languages to explicitly expose data communications. Such is the case of any streaming language.

Streaming languages support code portability by exposing compilers to the basic parallelism concepts, enabling both shared and distributed memory architectures support. Moreover and assuming the existence of sufficient optimizations, exploiting architecture memory model and static scheduling, one can take portability to yet another level and claim that streaming languages provide performance portability, assuming that compilers could interpret streaming languages and perfectly adapt code to the target system.

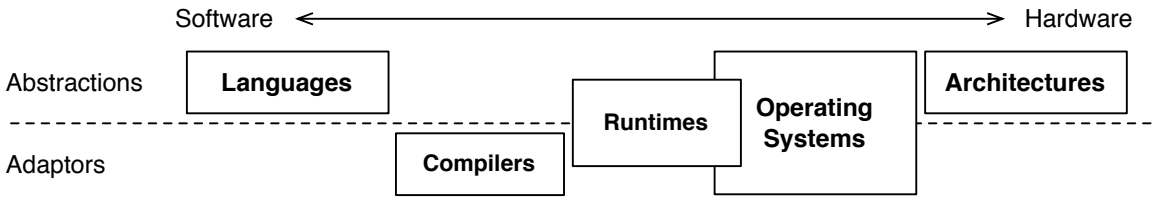


Figure 1.7: Computer systems abstractions and adaptors.

1.3 Computer System Adaptors

Computer systems are composed of several abstraction layers, each elevating its accessibility by hiding many of the complexities inherent to the lowest level abstraction (the architecture, in the context of this document). The chapter so far presented languages, operating systems and architectures as the abstraction layers. Each of these layers, when further analyzed and decomposed, might be more than just an abstraction. As an example, operating systems are indeed abstraction, but when further analyzed, one could detect that it is also an adaptor. The same operating system, creating the same abstractions, is many times implemented for different target architectures, being the code differences for the different targets, the adaptation part of the operating system. Moreover, an adaptor is a portability layer connecting the different system abstractions. The language and architecture layers can be understood as pure abstractions considering its highest and lowest positions in the hierarchy[1]. Every layer between the languages and architectures abstractions has at least some adaptability functionality and any change in the existing abstraction implies changes to the adaptation layers.

Adaptors are the components combining and adapting different abstraction levels. The abstractions are the running entities of a system, on the other hand, adaptors are only used through system preparation (compilation) and do not actively participate in the system execution environment. Compilers are an example of a pure adaptor entity. Other differences between an abstraction and an adaptor is its runtime system participation.

Figure 1.7 is a diagram differentiating the adaptor layers from the abstraction ones. Both types of components separate hardware from the highly abstracted world of software.

1.3.1 Compilation

For many years, compilers have been designed focusing its support on the still existing sequential programmable general purpose architectures. Biggest examples are the GNU Compiler Collection (GCC) [32] and LLVM [2], containing several intermediate representations and data structures spread through hundreds of optimizations phases.

High-level parallel languages cannot be optimized by this generation of compilers. To support such languages, its designers tend to create source-to-source transformers, translating languages high-level semantics into C code together with the language associated runtime library primitive calls. Such code is then compiled resorting to advanced production compilers (aka. mainstream compilers). Source-to-source compilers in most cases translate original language semantics into runtime libraries designed specifically for the language. The libraries

[1] In the context of this document, languages and architectures are not further decomposed and so can be understood purely as abstractions.

define the basic components of the language, highly optimized to explore the properties of target operating system and architecture.

However, because libraries are defined externally to the application compilation flow and due to its synchronization primitives implementation (containing volatile variables, low level synchronization primitives, barriers or mutexes), all of the language associated runtime calls are considered as possibly containing side effects. Side effects code by default disables code motion related optimizations of non pure code, i.e., disables the most relevant mainstream compiler optimizations, such as any loop optimizations (vectorization, loop invariant code motion) and partial redundancy elimination.

Source-to-source compilers, apart from implementing the language specific parallel optimizations, attempts to hide these code motion limitations by reimplementing most of the same optimizations already existing in sequential compilers. This approach reduces any sequential "high-tech" fully capable compiler into very basic translator. An exception to such design is the GCC's OpenMP [17] implementation, where its code lowering occurs in the front-end compilation stages. Unfortunately, its design suffers from the same weaknesses as source-to-source compilers, considering as its conversion occurs too early in compilation flow, creating similar external runtime library calls as source-to-source transformers and disabling many of the common existing compiler optimizations.

Without taking any merit from all of the high level languages, nor from the current generation of compilers, neither of these fields alone will, by themselves, be capable to solve the existing portability problems. For example, the problems introduced by the big distance and abstraction differences between existing high-level languages and architectures. To minimize the penalty for such gap, compilers must be able to lower higher level languages into closer to hardware representations, capable to cooperate with existing optimizations and further optimize parallelism and sequential code. Concurrency, synchronization and data communication are as important to parallel compilation, as functions, control-flow and data-dependencies are to sequential programs compilation. Without such properties integrated in a single compiler, it is impossible to perform the required analysis to optimize parallelism and the sequential code altogether.

Compilers must understand and unify both higher and lower level computer system abstractions, in order to adapt software to the hardware (architecture) properties and requirements.

1.3.2 Runtime and Operating Systems

Existing compilers are far from optimally supporting all the variations in computer architecture and micro-architecture. They typically make little use of the number of cores, processor interconnect differences, memory bandwidths and cache level sizes, not speaking about details of the micro-architecture. This diversity, through small variations in computer architecture properties, would turn targeted static compiler optimizations obsolete within matter of months. In other words, compilers cannot address such small architecture variations. To some extent, runtime libraries and operating systems can compensate and act as the adaptation layer, performing the small adjustments compilers should and cannot.

Albeit compilers cannot be held responsible for all degrees of application tuning, they can still gather precise application tuning information and provide it to target runtime system and operating system. This information is both directly collected from the high-level languages and as a result of compiler static analysis.

The architectures define the instruction set and lower level properties of a system and the operating systems define yet another abstraction layer. Depending on the actual device type the abstraction level can vary. For example, mobile devices by default contain very restricted characteristics regarding power consumption, when comparing with traditional desktop machines, and so its operating system should be rather simplified and contain fewer abstractions than a desktop environment.

For the particular case of parallelism, the relevant operating system abstractions are the threading model, thread scheduler and atomic operations. Depending on the actual available abstractions, the application programmer should take in consideration the target device and not only its architecture.

Runtime libraries, as previously mentioned, both abstract and adapt the compiler generated code to the target system. In case of streaming languages, runtime libraries might contain abstractions to buffer manipulation, task instantiation and synchronization. Moreover, runtimes also act as the portability layer (adaptor) for particular compiler code transformations. An example for such a runtime is *libgomp* which is the target runtime library for GCC OpenMP implementation.

Runtime systems often act as dynamic optimizers, adjusting system parameters based on static or dynamic system profiles. In some cases, runtimes, also based on profiling, can perform just-in-time (JIT) compilation, optimizing overall execution time. An example of such a framework is OpenCL (Open Computing Language) [4], which includes a JIT compiler to specialize compiling code for particular target during execution.

1.4 Wrap-Up

The large variety of parallel architectures and languages, together with all the inherent complexities introduced by parallelism, turns parallelizing compilation into one of the greatest enigmas for several generations of compiler engineers.

As time passes, more and more transistors are synthesizeable in a single chip. The higher the number of transistors per chip, the more architectures must be simplified in order to respect the power ranges for such miniaturized devices, forcing evolution to occur through parallelization and the replication of these cores. Multi-core architectures by themselves do not produce any performance boosts, as previous processor generations did and as consumers were used to. To get performance improvements from legacy applications, the compilers should evolve compensating for the now simplified and parallelized architectures.

Current generation of mainstream compilers are able to detect certain types of parallelism based on loop analysis, such as vectorization. However, in case of explicit parallelization either performed by programmers or source-to-source transformations, current generation of mainstream compilers are unable to interpret code as parallel, forcefully disabling any interfering optimizations.

Most well knowns programming languages (such as C), compilable through mainstream compilers, assume a single instruction stream and a monolithic memory, making these not very good candidates to express parallel execution models. On the other hand, newer parallel languages capture the (in)dependence and locality properties of an application, but are tightly coupled with there own compilers, hardly capable to exploit all of the available variety of parallel architectures.

Parallel stream and data-flow programming makes applications task-level and data-flow

explicit, exposing its pipeline, data and task parallelism while guaranteeing functional determinism. Unfortunately, such languages tend to be implemented through source-to-source compilers, helped with specific runtime library, performing the adaptation to the target system, but obfuscating mainstream compilers and disabling many of its optimizations. Both source-to-source and runtime libraries are tightly designed for specific target systems and high-level languages.

Mainstream compilers, although converting general purpose languages, must also understand the source-to-source lowered parallel code, allowing both parallel and sequential code optimizations. Moreover, as traditional optimizations transform the code, also the parallel code should be adapted and optimized by mainstream compilers. As an example, lets consider a source-to-source compiler performing a load balanced static scheduling based on the expected execution times of the compiling individual tasks. If any of these concurrent tasks is further optimized by the mainstream compiler, for example through vectorization, the predicted scheduling is no longer balanced and a new task scheduling must be done. The mainstream compilers must either be able to interact with the source-to-source compilers, identifying the right optimizations to apply, or be able to perform both parallel and sequential code optimizations.

Mainstream compilers contain several intermediate representations capable to represent sequential code semantics. Such representation by itself is not sufficient as it interprets the source-to-source generated primitive calls as external to the compiler knowledge and not as well known parallel primitives they might be. The streaming languages are between the best candidates for such an intermediate representation. Although, one must identify the right abstraction / expressiveness level. Too much abstraction leads to a less expressive language, reducing the variety of representable higher level languages. On the other hand, a too low-level abstraction will make code too expressive and impossible to analyze or optimize.

Together with the intermediate representation a new runtime library must be defined, expressing the intermediate language constructs and primitives, also acting as a second level adaptation layer for the language, guaranteeing the portability of the language in the similar target systems. Similar architectures with different cache level sizes can still share the same runtime implementation although using a different configuration. On the other hand, if the architectures have different memory model (shared or distributed memory), each must have its own different implementation. In any case, the compiler generated code for both these memory models is very similar.

The high-level languages should then be lowered into this intermediate representation either through their private source-to-source compiler, possibly performing a few optimizations before code generation, or inside the mainstream compiler at one of the lowering phases. Either way, any possible information collected during the conversion is stored in an independent data structure, defined to support decision by the intermediate language optimizations. Such is the case because the language certainly loses information during code lowering, considering the intermediate language is more expressive (lower level) than the original compiling language.

Well known mainstream compiler optimizations, such as dead code elimination and partial redundancy elimination, must be redesigned to provide further improvements in both the sequential and parallel code. Parallel optimizations, such as static scheduling, blocking, task fusion, among others, can also be implemented using such intermediate language, making use of collected information during the lowering phases.

1.5 Contributions

This thesis makes several contributions associated with streaming languages, parallel compilation, optimizations and streaming runtime implementations. The contributions are:

1. Erbium: a close to hardware and expressive streaming language, supporting multiple producers and multiple consumers communicating through a shared data structure abstraction.

2. A lightweight shared-memory runtime, with lock-free algorithms, supporting the dynamic construction and evolution of a graph of dependent, persistent processes. It implements all the language data structures and primitives. We also compare the design of this runtime library with alternative approaches, considering the target operating system and architecture properties.

3. An integration of the Erbium streaming language as an intermediate representation for mainstream compilers. It provides higher-level languages with a compilation middle-layer capable of avoiding semantical obfuscation induced by calls to runtime support library functions. A general purpose data structure collects the original language high-level properties during the front-end language lowering, using it later for compilation-time optimizations.

4. A study of the static analyzes available at the level of Erbium, much relevant for the design of task-level optimizations of a concurrent language and to the extension of existing compiler optimizations — such as dead-code elimination and partial redundancy elimination — to support parallel constructs. In particular, static analysis of Erbium allows to remove redundant synchronizations for general data-flow concurrency.

1.6 Thesis Outline

This chapter guided the reader through the parallelism-related abstractions, ruling the current generation of multiprocessors and presenting streaming computational models as a possible approach to bridge the software and hardware "worlds".

Chapter 2 introduces Erbium, a programmable, low-level streaming language, usable as an intermediate representation for compilers and the development of data-flow and streaming applications.

Chapter 3 presents a very efficient runtime implementation for Erbium, implementing the language primitives exploiting the x86 memory model and important microarchitecture properties of its memory hierarchy. This chapter also discusses alternative implementations for weak memory consistency models, as well as distributed memory architectures.

Chapter 4 presents an integration of Erbium as an intermediate representation within the GNU Compiler Collection (GCC).

Chapter 5 presents a possible compiler conversion from a streaming extension to OpenMP (Streaming OpenMP) into Erbium.

Chapter 6 is an overview of several compiler optimizations based on static analyzes of Erbium code, possibly supplemented by inter-procedural and inter-task properties whose static analysis is described in Chapter 4.

Chapter 2

Language

High-level parallel languages are highly abstracted from the hardware complexities, allowing programmers with enough tools to capture the application (in)dependence and locality properties. Compilers, on the other hand, are responsible for lowering these abstract languages to specific target architectures with well-orchestrated threads and

Streaming and data-flow semantics make task-level data-flow explicit, capable to expose pipeline, data and task parallelism while guaranteeing functional determinism.

Current generation of mainstream compilers, although not capable to express parallelism, have hundreds of very well orchestrated analysis and optimization stages.

In order to adapt data-flow programs, compilers must contain an intermediate representation, capable to express all of data-flow semantics and properties. Such intermediate representation would create an integration between legacy compiler optimizations, as well as provide enough analysis to perform parallel static optimizations and runtime instrumentation.

The Erbium language is defined not only to be such low-level streaming data-flow language intermediate representation, but also a low-level language for efficiency programmers. Its goal is to provide enough expressiveness to represent high-level languages semantics, preserving its compatibility with existing compiler representations, not obfuscating existing optimizations, but keeping the higher level language properties and rich abstracted information. Moreover, the Erbium language must provide traditional compilers with the ability to represent and optimize streaming data-flow languages.

Thanks to its streaming semantics it features a unique combination of productivity and performance properties, more precisely its determinism, expressiveness, modularity, statically adaptability and lightweight implementability.

Determinism

Erbium semantics derive from *Kahn Process Networks* (KPNs) [44]. KPNs are canonical concurrent extensions of (sequential) recursive functions preserving determinism (a.k.a. time independence) and functional composition. Functions in a KPN operate on infinite data streams and follow the Kahn principle: in denotational semantics, they must be continuous over the Scott topology induced by the prefix ordering of streams [44, 51]. Operational semantics of KPNs states that processes communicate through lossless FIFO channels with blocking reads and non-blocking writes. Although Erbium derives from KPN, preserving its determinism, its semantics are not bounded to the same operational implementation.

Expressiveness

Parallelism is often implicit in high-level data-flow languages [11, 46]. As an intermediate representation, Erbium defines explicit parallel constructs and primitives.

Erbium supports dynamic creation and termination of processes, favoring persistent, long-running processes communicating through point-to-point data streams.

Traditional streaming communications involve `push()` and `pop()` primitives over FIFO channels. Erbium's data structure for communication is much richer, providing random-access peek (read) and poke (write) operations, decoupled from the actual synchronization. Its decoupled synchronization allows to make dynamic adjustments to communication granularity, while preserving the process modularity.

Erbium's abstracting data structure is called an event record. It unifies streams and futures [38] and generalizes them to support multiple producers and multiple consumers.

Moreover, Erbium is able to hand-over, i.e., transfer consumption and production responsibilities between processes, dynamically redesigning the application process graph.

Compiler front-end or source-to-source compilers are responsible for the high-level languages conversion to Erbium.

Modularity

With the performance and power efficiency improvements from later architectures, the complexity of recent applications has also increased. applications are no longer the effort of a single individual but the collaborative work of big software development teams. Programming work-splitting implies languages to provide enough abstraction and development independence, i.e., modularity. In other words, language and compilation modularity is essential for the success of current generation languages.

When considering a language as an intermediate representation, functional independence or modularity is also essential. Traditional compilers through most of the compilation flow optimize code on a per function basis. Without modularity, compiler analysis would be too complex and optimizations nearly impossible.

An Erbium program is built of a sequential main thread dynamically instantiating concurrent processes [1]. Not only the main thread can instantiate processes but also any other function or process.

Since Erbium provides explicit means to manage resources (e.g., communication buffers), it puts a specific challenge on the ability to compose processes in a modular fashion. To this end, Erbium introduces a low-level mechanism for modular back-pressure supporting arbitrary broadcast and work-sharing scenarios.

Static adaptation

High-level languages are too abstracted from hardware properties not exposing many possible code optimizations. Binary code, on the other hand, cannot offer the required level of static adaptation. Erbium defines the intermediate representation as the portability layer between higher-level languages, target runtime systems and architectures. As an intermediate language, Erbium supports code specialization, static analysis and optimizations.

[1] Processes are not the traditional Linux processes but rather concurrent Erbium threads.

The compiler selects the most relevant hardware operations and inlines Erbium's split-phase communication primitives. It adapts the grain of concurrency through task-level and loop transformations.

The Erbium runtime may be transparently specialized for different memory models. Chapter 3 presents several of these runtime specializations, mostly focusing in a shared, global address space whose caches are kept coherent in hardware. The chapter also presents an efficient distributed memory approach for an Erbium runtime. In addition, Erbium may transparently exploit any hardware acceleration for faster context switch [8, 48, 75], synchronization and communication [35, 60].

Lightweight, efficient implementation

Erbium aims to be closest to the hardware while preserving portability and determinism. Any overhead intrinsic to its design and any implementation overhead hits scalability and performance. Such overheads cannot be recovered by a programmer who operates at this or higher levels of abstraction.

Thanks to its data-flow semantics, it is possible to implement the primitives of the Erbium runtime only relying on non-blocking synchronizations. Leveraging Erbium's native support for multiple producers and multiple consumers, broadcast and work-sharing patterns are implemented very efficiently, avoiding unnecessary copy in (collective) scatter and gather operations.

The split-phase communication approach hides latency without thread scheduling or switching overhead and relies entirely on existing hardware such as prefetching or DMA.

2.1 Related Work

Concurrency models have been designed for maximal expressiveness and generality [41, 55], with language counterparts such as Occam [22]. Asynchronous versions have been proposed to simplify the implementation on distributed platforms and increase performance [29], with language counterparts such as JoCaml [49]. Compared to these very expressive concurrency models, Erbium builds on the Kahn principle (data flow), which is sufficient to expose scalable parallelism in a wide spectrum of applications; it also offers determinism and liveness guarantees that evade the more expressive models.

Our work is strongly influenced by the compilation of data-flow and streaming languages, including I-Structures [11], SISAL [46], Lustre [37], Lucid Synchrone [20], Jade [73] and StreamIt [81]. These languages share a common interest in determinism (time-independence) and abstraction. They also involve advanced compilation techniques, including static analysis to map declarative concurrent semantics to effective parallelism, task-level optimizations and static scheduling. Erbium is an ideal representation to implement platform-specific optimizations for such languages.

The work of Haid et al. [36] shares many design goals with Erbium. It aims for the scalable and efficient execution of Kahn process networks on multicore processors. Its sliding window design matches the layout of records in Erbium. But it does not come with an associated compiler intermediate representation, and it does not deal with modular composition and dynamic process creation. Furthermore, although it advocates for streaming communications, it still relies on dynamic scheduling for event-driven synchronization. Most encouraging

to us, Haid et al. demonstrate that data-driven scheduling and streaming communications can coexist: it indicates that our approach is complementary to the large body of work in lightweight runtimes.

2.2 Semantics

The Erbium language with its very low-level (close to hardware) semantics and its intermediate representation defines a portability layer for high-level streaming data-flow languages, providing compilers with static adaptability and expressiveness not available at binary code level. The Erbium expressiveness comes from its decoupled representation, i.e., the language constructs partitioning into the three most basic parallelism components (thread creation, synchronization and data communication).

Thread creation is abstracted by the notion of *process*es. Processes are slightly more restricted then traditional threads, allowing compilers to have more opportunities to statically analyze Erbium applications.

Synchronization and communication are performed based on the notion of *events* and its *totally ordered* sets called *event records*. Records abstract and unify the notion of streams and futures [38], combining synchronization and data accesses into a single concurrent entity. The event is the abstraction for a single data element associated with a unique identifier. Views are the process interface to records, being the processes abstraction to synchronization and data communication.

The *view*s are thread locally stored data structures (defined within the process code) and connect processes with the multi-thread shared *event record* data structure. The *view* data structure also provides a set of thread-safe primitives hiding the synchronization and data communication complexities, associated with the concurrent accesses to a shared data structure such as event records. The primitives notify/query the *event record*, updating the local process *view* data structure state. The *view* thread local definition minimizes communication cost, by keeping a local status over global process progress, resulting in lightweight synchronizations.

The view and record data structures also make the Erbium language more *modular*. By separating the definition of shared data structure *event record* from *process* definitions, using the *view* data structure, programmers can independently design *processes* not taking into consideration how the process is used, i.e., its final application integration.

Views not only provide the means to synchronize and communicate data, but also to verify for resources availability through explicit request/release resource primitives. Such explicit resource management can be initially understood as spurious, considering that a reverse stream (back connection) allows to implement such verification without an implicit language support. However, the non explicit approach not only would break modularity, but would also lower the language semantical abstractions, converting resource usage prediction (buffer-sizing) into an inter-process analysis instead of only a single process analysis, i.e. resource management analysis would imply to analyze the connections and code of the respective processes.

2.2.1 Processes

Processes are the Erbium language abstraction to long-running and persistent concurrent function definitions. Although defined as long-running code entities, processes implementa-

tions are not tight to a particular scheduling model. The process instance threads can have either a user-level or kernel-level policy scheduling. Compiler transformations would be responsible to perform the necessary code adaptation to the target threading and scheduling library.

Processes are both defined and instantiated as traditional C function calls. Although defined as a traditional function, the processes cannot be executed as such. Processes can only be instantiated, i.e., executed in an independent and concurrent manner (in a thread).

The processes instantiation is dynamic and hierarchical, i.e., processes can instantiate other processes or even replicate themselves. The processes dynamic runtime properties supports greater application expressiveness and modular functional independence.

In Erbium, processes are the abstraction to threads, having no implicit dependencies with any other application code. Any required dependencies must be expressed through the synchronization/communication primitives. On the other hand, the main application thread waits for the termination of all instantiated processes right before the application termination. Although processes can be instantiated hierarchically, processes do not terminate hierarchically. Only the main thread implicitly waits for the process termination. Processes cannot tell if any of the other processes have terminated unless explicitly coded through the data-communication data structures (record and view data structures).

2.2.2 Record and View

Process synchronization and communication is abstracted in Erbium through the concepts of *event record* and *event view*.

The *record* can be understood as a shared and unbounded buffer of *events*. An event is an abstraction to a single data communication entity (token), indexed with a constant and unique integer identifier (*index*). Each *index* is associated to a unique *record buffer* position providing an indexed access to all of the *record events*.

Erbium records have shared buffer semantics where both producer and consumer can directly[1] access the *record events*. However, this semantics do not limit Erbium to support shared memory architectures, but rather abstract its data communication as occurring through a shared structure.

With its shared memory semantics and its decoupled synchronization and communication, Erbium supports peek and poke (read and write) data access patterns, allowing both consumers and producers to address record elements memory directly. Moreover, it supports multiple producer and multiple consumer patterns, not affecting process expressiveness and providing applications with the means to easily define broadcast, splitters and mergers communication patterns, without replicating its communication channels.

Although *records* are shared data structures and Erbium has shared data communication semantics, the *record* accesses are abstracted through the *view* data structure. Processes requiring access to a specific record must define a *view* and explicitly connect it to the record as a *reader* or *writer*.

A view can be understood as an events record sliding window, limiting the events safely accessible by its defining process (Figure 2.1). The view sliding window boundaries are manipulated by the process through a pair of view associated primitive functions, explained in Section 2.2.3. Views are defined with a constant *horizon*. The horizon should be defined

[1] Accesses to the record always occur through the view abstraction data structure.

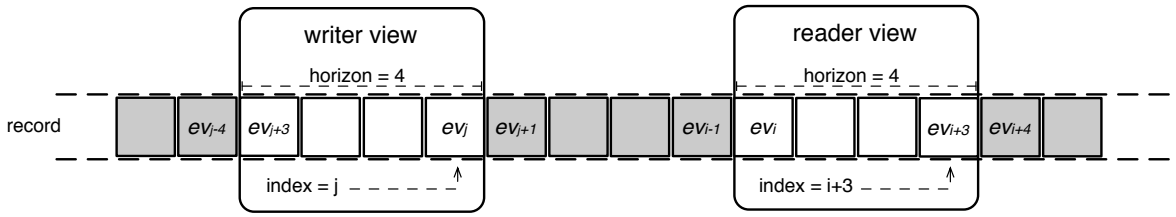


Figure 2.1: Record and view abstraction diagram. Views are sliding window over the record events. Events within the window are accessible by its defining process. Synchronization primitives move the sliding window and change access permissions to the record events.

has the minimum number of events required by its defining process algorithm (the number of simultaneously required record events). The horizon also limits the view sliding window from increasing above the horizon value, or in other words, it is the view sliding window maximum size.

Although not deeply studied thought this thesis, record resources size prediction (aka. buffer size prediction), can benefit from the process spread view horizons to define a minimum buffer size for the events record data structure. Optimal buffer-sizing not only avoids possible deterministic deadlocks (based on the lack of resources) but also produces better code execution times, hiding memory latency through double buffering.

2.2.3 Process synchronization and data communication

The Erbium synchronization and data communication primitives have decoupled implementations. Accessing events data does not implicitly performs a synchronization. The synchronization primitives however must be placed before and after events accesses. Doing so guarantees a correct and deterministic access to the record events data. This decoupled approach provides advanced programmers and compiler code generators with the tools to do granularity adjustments through loop optimizations, minimizing expensive overhead synchronization calls and adjusting code to the target architecture specifics such as its cache memory sizes.

The synchronization primitives define the events that are deterministically accessible thought the view and, depending on the type of view, "enforce" a specific type of allowed events access.

Synchronization and event state mutation

Erbium synchronization can also be interpreted as the management for the *record events*. Each event, through its "life", traverses five distinctive states — *Not allocated*, *Writable*, *Ready*, *Readable* and *No longer used*) — each classifying the type of access permitted by its connected views or processes.

The *event* state changes develop from the usage of the four Erbium synchronization primitives (*update*, *release*, *stall* and *commit*) contributing to a specific change in the events state and consequently to inter-process synchronization. Primitives are called in the process code, each call specifying in its arguments both a view and an event index. The view and the index arguments, through its connectivity, specify which record and which set of events are affected and should suffer a state change. Every event with an index smaller than the

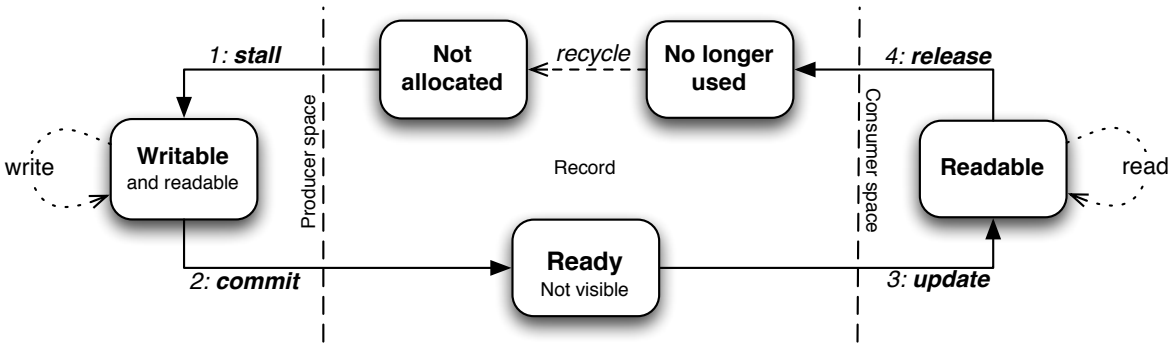


Figure 2.2: Events life cycle state diagram.

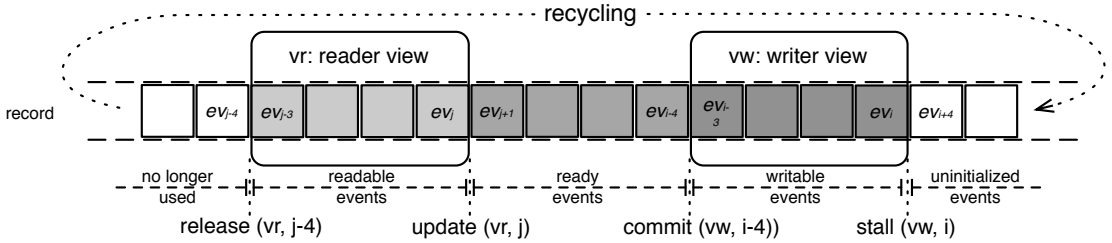


Figure 2.3: Local view visibility of events based on synchronization primitive calls in a reader (left) and writer (right) view example. The diagram shows the different event states in the horizontal dashed line regions and its relation with the synchronization calls below. Dark grey events are writable by writer view, light-grey are the read-only through the reader view and mid-grey events are ready events that are still not accessible.

provided index argument is affected, unless it has already been affected in a previous primitive execution.

Figure 2.2 layouts the five different states of an event life cycle and the primitives responsible for each state change. Moreover, the different primitives are only executed in specific process contexts. *Stall* and *commit* are called only using *writer views* (producer processes), while *update* and *release* are called through *reader views* (consumer processes). Every allocated event traverses all the five states and each state transfer only occurs with a specific primitive call type. Please realize the existing dependencies[1] between those state changes and consequently between the different primitive calls. Dashed lines divide the events between producer, consumer and shared memory spaces contexts.

Respecting such dependencies is fundamental to guarantee Erbium applications determinism. Also, all the synchronization primitives are dependent, based on the events state diagram, not all of these dependencies must be enforced through runtime support. Stall and commit, as well as update and release are always executed within the same process context and so sequentially executed. Static validation for such dependencies is a plus and, although not studied in the context of this thesis, can be deduced based on the analysis explained through Chapter 6.

Figure 2.3 is a birds eye of the record events buffer, showing the record events, its state, two views (reader and writer) and the primitive calls, enforcing the presented events states.

[1] These dependencies are later used in Chapter 6 in the context of the Erbium intermediate representation analysis and optimizations.

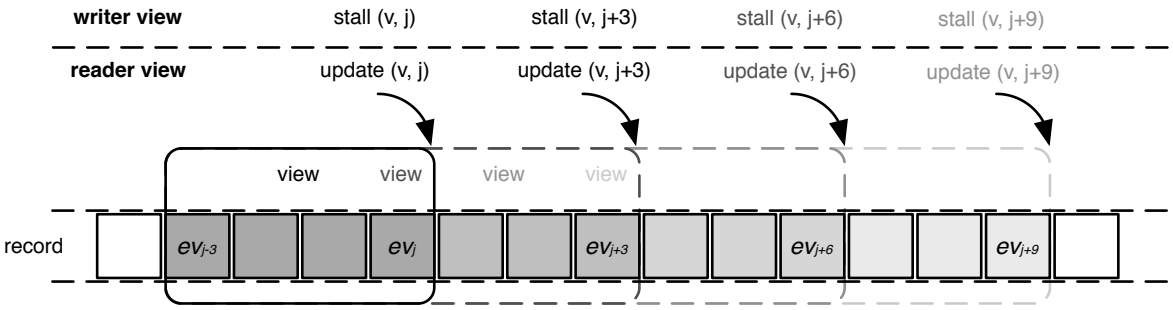


Figure 2.4: Local effects of *stall* and *update* primitives. $stall(v, j)$ and $update(v, j)$ define the view position and visibility of its record events. A call to *update* or *stall* moves the view up to the event with index j. Events out of the view boundaries are inaccessible by its owning process.

Each of the squares if the diagram represents a single event tagged with its index identifier. The queue of events represents the record data structure. Each of the views is associated (defined) to a specific process not represented in the diagram, being the views of either a reader or a writer of the record. Events inside of the view sliding window (round cornered boxes) are accessible by the respective view associated process. The primitive calls specify the presented sliding window bounds and define which events are deterministically available to each view and associated process.

View sliding window manipulation

In order to better understand data accessibility it is important to understand the impact synchronization primitives have on the *view* visibility over its *record events*. Depending on the usage of the synchronization primitives, each process controls which events are deterministically accessible within the view sliding window.

A $stall(v, j)$ or $update(v, j)$ primitive call represents a requests from the calling process to expand the view sliding window visibility, such that record events up to the index j become accessible. As these primitives extend the visibility of the view, it is possible that the primitive blocks the process execution when no events are available, i.e., the requested events are not in *ready* state, in the case of the *update* primitive, or if no free resources available, for the *stall* one. In other words, *update* and *stall* control the head of the view sliding window and has they request for newly events to become available such primitives might have to block. When blocked, lower level inter-process synchronizations verify for any concurrent record *commit* or *release*, unblocking and expanding the view sliding window in case enough events become available. When unblocked, the process is guaranteed a deterministic access to the primitive associated events, writing or reading its data as presented in Figure 2.3. Figure 2.4 shows the impact of *stall* and *update* on the view visibility over its connected record, when different "flavors" of calls are executed. Please notice the semantical similarities of *update* and *stall* primitives.

Contrarily to *stall* and *update*, *commit* and *release* collapse the view sliding window size. By calling *commit* or *release*, the process is specifying it no longer requires to access any events below the argument provided index. When executed, the calls notify the record that the affected events are no longer accessed by the associated process and the events are updated with the respective state change.

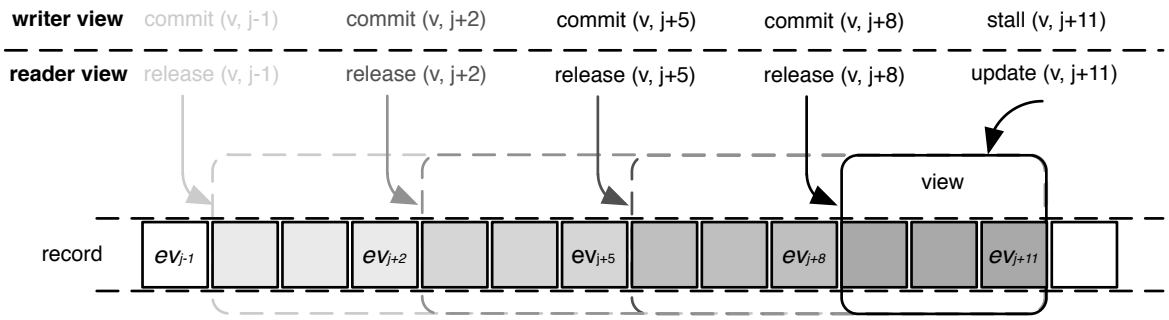


Figure 2.5: Local view effects of *commit* and *release* primitives. These primitives signal the record dependent views to proceed. This signal occurs after reducing the internal access privilege to the respective events (the sliding window lowest bounds). After executing the specific *commit* or *release* primitives (on top), the view sliding window is shrunk and the event to the left of the dashed line becomes unaccessible.

Figure 2.5 is a diagram of a view to which several commits or releases are executed for a single previous *stall* or *update* call, respectively. One can realize that both *commit* and *release* control the tail (lowest bounds) of the view sliding window.

View events data access

Erbium borrows data access syntax from widely known C arrays (however using double square brackets). Any view is usable if as it was an array, accessing the events in its connected record. For example $v[[i]]$ is a direct access to the event data in the record connected by v and identified with the monotonic index i. Compared to arrays, which expect the usage of an index within the array bounds, views expect an *event index* (monotonic unbounded index).

As previously mentioned, each record contains a buffer of events. This buffer is not strictly defined by the Erbium language, as different architectures might require different buffer implementations. Instead, buffers are implemented at the language runtime support level. Independently of buffer implementation, the Erbium code has portability as a guarantee. This is the case since event accesses are expanded by any Erbium supporting compiler, reducing the overhead of these accesses to its minimum. There is a wide range of possible record event buffer implementations, such as circular buffers or dynamic growing buffers, possibly implemented through traditional arrays or indexed linked lists. Through Chapter 3, further detail is given regarding possible buffer implementations and its minimal requirements for Erbium's integration.

Moreover, in a distributed memory architecture, it is not possible to have a single shared buffer but instead many distributed ones. Albeit that, Erbium applications are implemented assuming a single shared buffer and as long as event dependencies are respected, the compiler lowering and runtime implementation must guarantee a valid and deterministic result.

As data accessibility is decoupled from synchronization, it is entirely the responsibility of the programmer or compiler to introduce the necessary synchronization calls in order to preserve data accessibility determinism. Static compiler analysis or debugging runtime libraries can also be implemented to detect possible incorrect and non deterministic Erbium applications. The possible Erbium static analysis and optimizations are discussed in Chapter 6.

The inter memory space data transfers can also be requested explicitly. This is done

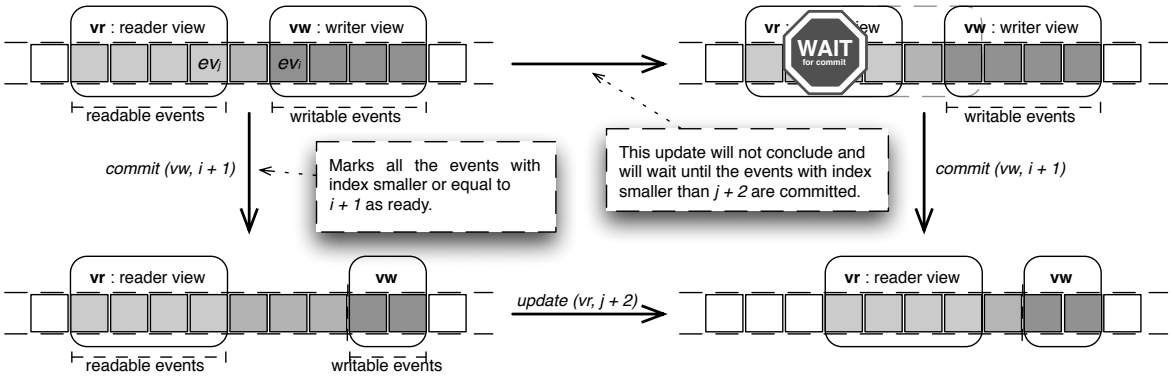


Figure 2.6: Synchronization scenarios between commit and update.

through the *transfer* primitive. The primitive allows the programmer to anticipate which events are necessary for the upcoming process iterations, performing the requested operations and moving data from the different processes memory spaces. Each process independently calls transfer primitives, notifying the record of the precise events the process produces and consumes. Once the events are *committed* or *released*, an asynchronously transfer call is immediately scheduled. This approaches boosts performance by hiding memory latency, improving the data locality by anticipating future process data dependencies.

This primitive has a polymorphic implementation allowing the request of all the events up to a specific index or a range of events (pair of indexes).

- **transfer** $(view, i)$ - initiates as soon as possible an asynchronous transfer of the events up to the event with index i to the local memory (local cache) of the processor executing the process associated with $view$.
- **transfer** $(view, li, ui)$ - initiates a similar transfer but only transferring the events between the provided index boundaries. This version allows to reduce the transfer cost in cases where only a subset of the events is required.

Moreover, this primitive is implemented through Erbium runtime support and is defined based on architecture memory model and the available asynchronous memory transfer instructions. Details on implementation for both shared and distributed memory architectures are presented in Chapter 3. As this primitive is redundant in shared memory architectures, considering the existing memory transfer abstraction (caches), most of the examples available in this document do not make use of the *transfer* primitive.

Synchronization example

Consider a simple producer and consumer application containing processes P (producer) and C (consumer). During its instantiation both are provided with a record r in its arguments and each connects its own private view (vw for P and vr for C) to r.

After several iterations of both processes execution, the record and views get to a point similar to the one presented in Figure 2.6. The multiple paths presented in the diagram result from both processes concurrent execution. Nevertheless and independently of the path, the outcome of the execution is the same or determinate.

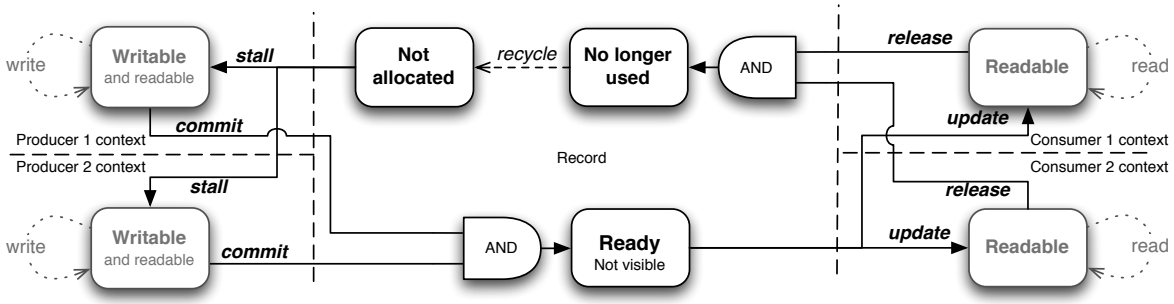


Figure 2.7: Events life cycle state diagram on a multiple producer multiple consumer record.

As presented in the diagram, each of the process has an induction variable (i and j) and at the top-left diagram $i = j + 2$. At this point, as we have only two concurrent processes, either P is the faster and the *commit* primitive is executed first, or C is and *update* is executed. In case C is faster, the does not returns and it blocks, waiting for the dependent *commit* to occur. Once the *commit* is executed the *update* returns and the consumer process proceeds its execution. Once P executes the *commit* and it is perceived by C, the update unblocks and C can continue its execution. Please notice, at the upper-right case, that both the reader and writer view sliding windows are overlapped in the case where the update should block. If the *update* primitive did not block, it would result in a non deterministic access to all the overlapping events.

At any time, events present in a view sliding window can be written or read by the view owing process (thread). The synchronization primitives guarantee deterministic access, disallowing events from simultaneously being read and written from different processes. Please notice that once an event changes state such event can never rollback to its previous state (Figure 2.2).

2.2.4 Multiple producer and multiple consumer

Although the previous details focused attention to a single producer and consumer design, Erbium supports multiple producer and multiple consumer communication and synchronization patterns. This section focuses attention on the language design concerns for multiple producer and consumer applications, part from a deeper discussion of the event dependencies while preserving the language determinism for such types of communication.

Figure 2.7 layouts a similar event state diagram as presented in Figure 2.2. As the previous figure focused in a single producer and consumer example, not all the dependencies were exposed. This new diagram is split in five different contexts referring to two producers, two consumers and the respective connected record shared memory space. Considering the Erbium shared semantics, each producer and consumer is able to access exactly the same events. Without extra logic, determinism would be in jeopardy in multiple producers or consumer applications.

What should happen when only a single connected view commits?

As synchronization does not necessarily involve writing the event value, it is mandatory that all the producers or consumers *commit* or *release* before the event can be considered *ready* or *no longer necessary*, respectively. Moreover, although an event can be shared within multi-

ple processes, in the context of a view, only itself is accessing such events. In a global context, the record knows all the connected views and which events are possibly being accessed. The record waits for every connected processes to access and dismiss any event, through *commit* and *release*, before any further progress is observed by dependent operations, more precisely by *update* and *stall* primitives.

The diagram in Figure 2.7 enforces commit or release dependency by including an "*AND*" within the transactions from the multiple concurrent "writable" or "readable" states into the record context "ready" or "no longer used" states.

If all the events are shared by concurrent processes, what is the consequence of accessing the same event?

All the *record events* are shared between the connected writer or reader view types, i.e., such events can be written or read by multiple processes. Read accesses do not impose a problem, considering that most architectures perfectly support concurrent multiple reads. On the other hand, if multiple processes write to the same event, either the event data becomes a mix of both writes in case of more complex data structures or only one of the writes would be effective. Moreover, the result of multiple writes to the same event has an unpredictable outcome, or in other words, a non deterministic result.

Erbium language advocates for disjoint multiple producers, meaning that each of the producers, when instantiated, is provided with enough information to decide which events it should be writing to.

The *transfer* primitive is an essential part of multiple producers and consumers, specially when considering distributed memory model architectures. Such is the case because in these architectures the record of events (buffer) is distributed through the memory spaces local to the processes. The *transfer* primitive usage allows to unify the many distributed buffers, providing the necessary information to combine all the distributed buffer into a single unified one. This topic is further studied and explained in the distributed memory section of Chapter 3.

How does multiple producer and/or consumer records impact views/processes?

Semantically, multiple producer and consumer records have no significant impact on how views and processes are defined. Moreover, from the process code perspective it is impossible to predict the number of record connecting views.

In multiple producer and consumer records where each of the processes accesses only a subset of the events, all the non required events should be, as soon as possible either *released* or *committed*. This strategy guarantees a faster progress for its dependent processes, by anticipating event state progress for all of the process non dependent events.

As an example, consider two producer processes, each writing in groups of 10 events. If the processes only commit by the index of its last produced event, the consumer data availability is decided by the rate of the earliest producer process index. On the other hand, if each process commits until its next required event, record progress is perceived earlier, resulting in much faster synchronizations. Figure 2.8 presents possible execution traces of the two presented scenarios. The version on the left delays the promotion of events into "Ready" state since the process is only committing by the event it last accessed (wrote). The right version anticipates commits of undesired events, avoiding global record progress to occur at the rate of the smallest event index. The reader views or consumer processes must do the same, although in this case using *release* primitive.

P_1	P_2	Ready
$v_1[[1..10]] = \cdots$	$v_1[[11..15]] = \cdots$	< 0
commit$(v_1, 10)$	$v_1[[16..20]] = \cdots$	< 0
$v_1[[21..30]] = \cdots$	**commit**$(v_2, 20)$	< 10
commit$(v_1, 30)$	$v_1[[31..40]] = \cdots$	< 20
$v_1[[41..50]] = \cdots$	**commit**$(v_2, 40)$	< 30

P_1	P_2	Ready
$v_1[[1..10]] = \cdots$	**commit**$(v_2, 10)$	< 0
commit$(v_1, 20)$	$v_2[[11..20]] = \cdots$	< 10
$v_1[[21..30]] = \cdots$	**commit**$(v_2, 20)$	< 20
commit$(v_1, 40)$	$v_2[[31..40]] = \cdots$	< 30
$v_1[[41..50]] = \cdots$	**commit**$(v_2, 40)$	< 40

Figure 2.8: Traces of two possible implementations executions of multi-producer processes. Unoptimized version (left) delays the promotion of the events into "Ready" state. The optimized version (right) anticipates the commit of the events not accessed by the process. Two left-most columns are the executed operations for P_1 and P_2. The right-most column is the index of the latest record *ready* event.

In a multiple producer and consumer record,this is the case because events are only considered "Ready" or "No longer used" if all of its writer and reader views have *committed* or *released*, respectively. Like previously mentioned, *update* and *stall* are the primitives blocking and waiting for such conditions. If for some reason, one of the processes participating in a multi producer or consumer record do not *commit* or *release*, the application progress is at risk, resulting in deadlock.

Code complexity of both single and multiple producer and consumer records is discussed in Chapter 3, together with its implementation details.

2.2.5 Initialization and termination

Initialization and termination are common sources of complexity and deadlocks in concurrent applications. Erbium is no exception and precise attention to details is necessary, in order to overcome these complexities.

Erbium's process initialization and termination are dynamic and explicit. Initialization occurs through process instantiation, while termination occurs only as a consequence of process function termination. Both initialization and termination are explicit and on the demand of the application programmer or compiler code generator.

Before any process is instantiated, the communication necessary records must be allocated. Records should be provided as arguments during process instantiation. The process code explicitly allocates views, later connected to the records that are provided as arguments. Once connected, processes use the view to call synchronization primitives and access events.

Although by the current explanation initialization seems straightforward, there are a few omitted details complicating this pattern, more precisely, multiple producer or consumer communication.

How does the record knows how many views must be connected, before it notifies any reader view for ready events?

Previous explanations only consider single producer and consumer records. In order to maintain Erbium determinism, it is necessary that a record knows the initial expected number of connecting views and waits for such number of connections, before any progress is announced to reader views. For example, while not all of the writer views are connected and have committed, no progress can be predictable by the reader views. Similarly the same has to occur with reader views. While not all consumers are connected, no release primitive call should change the state of the events and no events are recycled (stall is eventually blocked).

Let's assume two producer processes similar to the ones presented in Figure 2.8. In this case, assuming P_1 is executed in a much slower CPU and that P_2 is able to connect and commit before P_1 is still being instantiated, it would be possible that a consumer could connect and update before P_1 is able to even connect. As process P_2 does not write at the indexes in the range 1..10, the events would be uninitialized.

In order to avoid such non-deterministic cases, it is necessary to tell the record the number of views planning to connect. Erbium language defines a primitive to specify this initial number of reader and writer views. Unless at least that specified amount of views is connected, the record progress is halted.

View allocation and initialization is non blocking. If for some reason, for example architecture properties, initialization must wait for all of the record connecting views, such wait should only occur at update or stall primitives. *Update* and *stall* should be the only two blocking primitives in Erbium. In a shared memory implementation (as presented in Chapter 3), once a producer process has its views connected to the respective records, the process can immediately start executing and writing events, even if the record is still waiting for other connections. On the other hand, consumers depend on the connection and *commit* of all the record expected writer views.

Moreover, *update* primitive blocks while not all of its record writer views have connected and *committed*. Similarly, *stall* blocks if no sufficient resources are available. However, after initialization, the resources are always available and producers should never be blocked during initialization.

What about termination, how do consumers detect termination?

After processes have no more work to do (events to produce), they should declare their intention to *free* their defined views, disconnecting them from the record. Once all writer views get disconnected from a record, it is classified and flagged as zombie. Zombie records do not allow further view connections, i.e., no more events are ever committed to zombie records.

Once update is called on a zombified record, the update return value allows to identify if the view connected record is a zombie, providing the process with enough information to gracefully free its views and terminate. Once the last view connected to a record (reader view) is disconnected (freed), the record data structure is deallocated. Moreover, a zombified record cannot be "brought back to life", i.e., it no longer can be connected by new writer views. "Resurrecting" a record would result in non deterministic semantics.

2.2.6 Process hand-over

The Erbium applications not only can benefit from dynamic process creation, but as well from dynamic communication network adaptation. These applications involve not only the process dynamic creation and termination but also to transfer process responsibilities, i.e., the views of the upcoming process instances must be able to take the place of the terminating ones, or as we call it to hand-over.

Erbium semantics supports such kind of dynamic communications. Any process hand-over involves disconnecting (freeing) at least one view from the record and allows some other process (view) to connect in its place and to continue its work.

In single producer single consumer examples, such problem might seem trivial and requires no extra complexity, mostly because one can assume that if any of its views stops committing

or releasing, the dependent process execution is halted, leaving no chance for data changes to occur until the substituting view is reconnected. However, in a multiple producer or consumer application, if a process is disconnected, the Erbium semantics guarantee the substituting view is able to continue the work of the previously disconnected one without any lost events. In other words, the newly connected view should have the same events in its sliding window as the disconnected view had.

However, not always such hand-over constraints are necessary, more precisely if the process does not need to continue the previous process work but rather only access the record content from that point on. Such processes by definition have a non deterministic initialization, considering its inability to block the record progress, i.e., the initial view sliding windows cannot be guaranteed to contain any precise range of events.

Considering both scenarios, two types of view connections are supported:

- **Registered** connected views are identifiable by the creation of a view connection id. Such id will allow other processes and views to connect as the substitutes of a previous connected one. View termination in such cases specifies if the view identifier should "die" with the disconnection or if other process is supposed to reconnect using this particular view identifier,

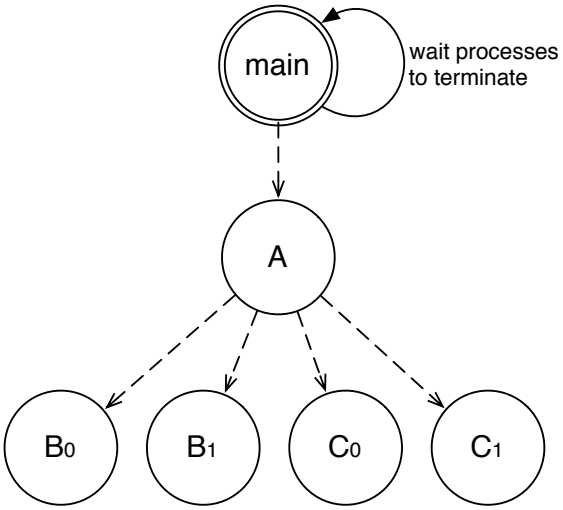
- **Non-registered** connected views, although having deterministic properties while connected, do not support process hand-overs. When disconnected from the record, no information of its state is preserved.

Considering the newly registered view connections, it is necessary to disambiguate process hand-over view disconnections from termination related ones. To express the two possible scenarios, two "flavors" of view disconnection must be supported:

- disconnecting the view while keeping the view connection id, not allowing the record to become "zombified" and permitting other processes to reconnect using this same connection id,

- disconnecting the view and deleting the connection id, resulting in the last connection using that same identifier, later resulting in the record termination (zombification).

As hand-over decision occurs at view disconnection, it is the responsibility of the process to determine if a particular view is either being disconnected for a process hand-over or permanently terminated. Moreover, when freeing a view, a process decides if any future processes will attempt to connect to the record using the same view connection identifier.




```
1  int main() {
2    RUN A(...);
3  }
4  PROCESS A (...) {
5    for(int i = 0; i < 2; i++) {
6      RUN B(...);
7      RUN C(...);
8    }
9  }
10 PROCESS B (...) {
11   ...
12 }
13 PROCESS C (...) {
14   ...
15 }
```

Figure 2.9: Process creation example. The diagram on the left shows with dashed lines the different process instantiations coded in the example on the right. The presented hierarchy only represents the process instantiations and its dynamic behaviour and not any type of hierarchical dependency existing between the process instance creator and the instance.

2.3 Syntax

Like previously mentioned, Erbium processes are defined like traditional C functions without a return type. However, process functions are prefixed with the *process* keyword guaranteeing that compilers can distinguish and validate its usage. Please remember that process functions cannot be called as traditional functions, but only instantiated (executed) concurrently.

- **process name (type_1 param_1, ...) { ...code ... }**

Like any other regular function, processes can be defined containing any type and number of parameters, including variable number of parameters.

Process instantiation is also very similar to traditional C function calls, although prefixed with the keyword *run*.

- **run name (arg_1, ...);**

During compilation, the necessary validations are executed, guaranteeing the instantiated functions are in fact processes, i.e., functions prefixed by the *process* keyword. The compiler transformation lowers the presented syntax into a target architecture specific runtime support library, later presented in Chapter 3. One such transformation is, for example, to convert both process definitions and instantiations into single parameter function. The transformation performs marshaling of all the process parameters into a single data structure. Typical thread support libraries, such as the POSIX threads API, by default enforce single argument thread functions.

Figure 2.9 code example illustrates how to define and instantiate processes right next to the graphical representation of the coded instantiations. In this example, three hypothetical processes (`A`, `B`, `C`) are defined. Process `A` is instantiated by the `main` function, while `B` and `C` are instantiated twice from the process `A`. Once the process `A` creates all the instances

```
1  PROCESS Producer(record int r)
2  {
3    view int view;
4    INIT_VIEW(view, WRITER, 1);
5    CONNECT_REGISTERED(view, r);
6    // STALL -> WRITE EVENTS -> COMMIT
7    FREE_VIEW(v);
8  }
9
10 PROCESS Consumer(record int r)
11 {
12   view int view;
13   INIT_VIEW(view, READER, 1);
14   CONNECT_REGISTERED(view, r);
15   // UPDATE -> READ EVENTS -> RELEASE
16   FREE_VIEW(v);
17 }
```

```
18 int main() {
19   record int rec;
20
21   add_registered_views(rec, 1, 1);
22
23   RUN Producer(rec);
24   RUN Consumer(rec);
25
26   return 0;
27 }
```

Figure 2.10: Record and process views initialization and termination.

of `B` and `C`, it will immediately terminate. Like explained before, processes do not contain any explicit dependencies. On the other hand, the application main function contains a list of all the instantiated processes and before terminating, the main threads waits for all process instances to terminate. Moreover, Erbium applications only terminate once all the instantiated processes have terminated.

Inter-process dependencies are always defined through *records*.

Records

Before the application instantiates any inter-process dependent processes, it must allocate and initialize the required records.

- **record T r**, defines a new record `r` whose events store data elements of type `T`.

The type of the record (T) must be any valid C language type. $Void$ typed records, or records that would not transfer any data are not valid. The reasons for such design are related with Erbium primitive relation dependencies and its optimizations, as explained in Chapter 6.

The record data structure initialization and buffer allocation are implicit to the record definition and converted to the target runtime support library during compilation. The record buffers are lowered during compilation and depend on a final buffer implementation for the specific architecture/application.

After its definition, the record should be set with its initial number of registered consumers and producers.

- **add_registered_views(record, nr_readers, nr_writers)** sets the number of expected registered reader ($nr\_readers$) and writer ($nr\_writers$) views.

Views

Views are always defined and initialized inside of processes, being always associated to that process and can never be transfer, i.e., must always be used in the context of its defining process. Moreover, record accesses are always abstracted by the view data structure.

- **view T v** defines a view *v* capable to connect to records of the same type (`record T`).

Before any process is able to communicate, it must first initialize its defined views and connect them to their respective records.

- **init_view(view, type, horizon)** specifies the *view* type and horizon size, later used by compilers in static analysis and compiler optimizations.
- **connect(view, record)** connects *view* as a reader or writer (type provided at view initialization) to the provided *record*.

In order to perform a registered connection, the record is first queried for a view connection identifier through:

- **view_id get_new_view_id(record, access_type)** which provides a view connection identifier for the record.

Once the *view_id* is obtained:

- **view_id connect_registered(view, record, view_id)** connects *view* to the *record*, registering it with the provided *view_id* (view connection identifier) and returning the connection identifier (*view_id*). If the connection identifier parameter is omitted, a new view connection identifier is created and returned from the function. The new identifier is obtained through an implicit call to `get_new_view_id`.

After the connection, the view is initialized and prepared to for regular usage by its defining process. Once a process has finished using a view, it should call:

- **free_view(view, id_terminate)** disconnecting *view* from its previous connected record. If *id_terminate* is *true* its associated view connecting identifier should be removed, in which case no other view (process) can connect to the record using the same identifier, i.e., perform the hand-over of the disconnected view. Once disconnected, views cannot be re-connected to any record.

A process may request any of its views for deallocation through `free_view` primitive. This primitive enforces no future view usages and is both used for view and record termination or even to initiate an hand-over. In case of a hand-over, it is guaranteed that the reconnecting view maintains the current sliding window bounds as previously disconnected view. The parameter `id_terminate` specifies if the view connection is permanently disconnecting or the `free_view` is part of an hand-over.

When all the writer view connection identifiers are removed from the record, it is considered as terminating, cascading termination to all the consumer processes. Once terminating, the record rejects any future writer view connections.

Figure 2.10 is a code example for the process instantiation and record / view initialization and termination, exemplifying all of the previous mentioned Erbium language constructs.

Process synchronization

As explained in this chapter, data communication and process synchronization are explicit and exposed by the view data structure, its synchonization primitives and the events accesses.

The following synchronization primitives are available in Erbium:

- **int update(view T v, int i)** waits for all the events with index smaller than `i` from the view `v` connected record. Its return value is always `i` unless all the writer views have

requested to terminate, in which case the record is zombified, i.e., will never contain an event with the index `i`. In this case, the update primitive returns the biggest event index available ("in *ready* state") accessible by the view `v`.

- **void stall(view T v, int i)** has similar semantics to *update* but is only used within producer processes checking for resource availability. Unlike *update*, *stall* does not return any value.

The commit and release primitives allow the process to announce which events are no longer written or read, respectively.

- **void commit(view T v, int i)** tells its view that the events up until index `i` are no longer modified (written) by the calling process (the view defining process). Moreover, this primitive signals the record reader views (consumer processes) of the availability of the now committed events. Processes should never access the committed events, because doing so breaks the application determinism. If this process requires to read past committed events, it must explicitly define a new reader view, connecting it to the same record.

- **void release(view T v, int i)** is symmetrical to commit but instead allows a consumer process to specify that the events with index smaller than `i` are no longer used by the view and its resources can be reallocated to bigger index events.

Figure 2.11 is an example of a producer consumer application only showing the synchronization related code. The diagram on the right is the expected control-flow graph from each of the processes (full lines) and the synchronization primitives dependencies (dashed lines).

These primitives apart from synchronizing concurrent processes also manipulate the views sliding window allowing processes to deterministically access the record shared events data.

Record events access

In Erbium only the events within the owned view sliding-window are accessible to processes.

`T& View::operator [[]] (const uint index)` is a pseudo definition of the events accessor for a view with definition `view T v`. This definition allows right hand side usage (read) as well as left hand side usage (write). Read or write usage of this operator is dependent on the access type of the view. Writer views can write or read from any of the events already stalled and not yet committed. On the other hand, reader views can only read events that have been updated and not yet released.

Notice the following examples:

- `v[[i]] = var;` assigns the value of variable `var` to the event accessible through view v with index `i`,

- `var = v[[i]];` assigns the content of the event `v[[i]]` to the variable `var`.

The usage of this operator is guaranteed as valid, as long as the events accessed are within the view (v) sliding window.

```
1  PROCESS Producer (record int r) {
2    view int v;
3    INIT_VIEW(v, WRITER, 1);
4    CONNECT_REGISTERED(v, r);
5
6    for(int i = 0; i < 2; i++) {
7      STALL (v, i +1);
8      ... algorithm events access ...
9      COMMIT (v, i +1);
10   }
11   ... termination ...
12 }
13
14 PROCESS Consumer (record int r) {
15   view int v;
16   INIT_VIEW(v, READER, 1);
17   CONNECT_REGISTERED(v, r);
18   int i = 0;
19
20   while(UPDATE (v, i +1) &=& i +1)
21   {
22     ... algorithm events access ...
23     RELEASE(v, i +1);
24     i++;
25   }
26 }
```

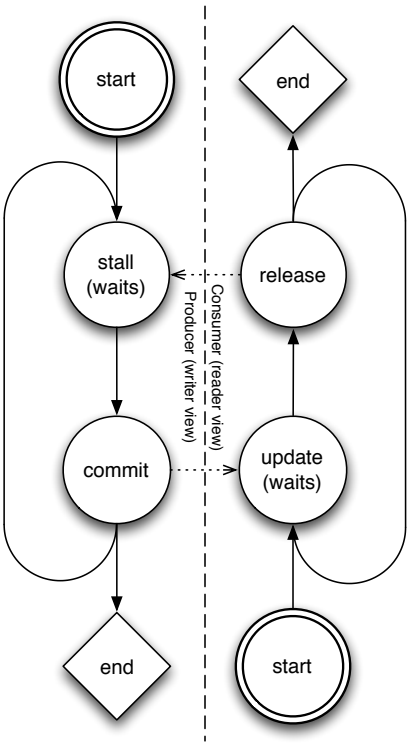


Figure 2.11: Producer-consumer synchronization code example and communication diagram. The code represents a producer process performing ten loop iterations. At each iteration it requests a new event (*stall*), performs its computation, and *commits* by the same index (the same event). The consumer iterates within a `while` loop, waiting for newly events (*update*). Once the producer stops committing, *update* stops returning bigger indexes and the while loop terminates. At each iteration, the *release* is called, releasing previous required events and notifying any blocked producer *stall* call. The diagram is a simplified control flow graph of the example on the left. Solid lines are control flow dependencies while dashed lines are the dependencies enforced by the Erbium synchronization primitives (*stall*, *commit*, *update* and *release*).

A view provides an abstraction for accessing events in a record buffer. This abstraction does not enforce any type of buffer implementation as long as it is possible to match an Erbium monotonic index to a buffer element (memory position).

2.4 Use cases

The previous section presented the Erbium language syntax, together with some small code examples of the language constructs and primitives. This section presents more advance Erbiums use cases, demonstrating the language properties and expressiveness, briefly comparing with other higher level streaming languages.

The entry point "Hello World" Erbium application is a single producer and consumer process communication, as presented in Figure 2.12. Both the producer and consumer processes communicate through a record of `int` typed events, defined externally to this example, similar to what was previously presented in Figure 2.10.

```
1  PROCESS Producer(record int r)
2  {
3    view int v;
4    INIT_VIEW(v, WRITER, 1);
5    CONNECT_REGISTERED(v, r);
6
7    for(int i = 1; i <= 1000; i++)
8    {
9      STALL(v, i);
10     v[[i]] = i;
11     COMMIT(v, i);
12   }
13
14   FREE_VIEW(v);
15 }
```

```
16 PROCESS Consumer(record int r)
17 {
18   view int v;
19   INIT_VIEW(v, READER, 1);
20   CONNECT_REGISTERED(v, r);
21
22   int i = 1;
23   while(UPDATE(v, i) == i)
24   {
25     printf("Hello World: %d\n", v[[i]]);
26     RELEASE(v, i);
27     i++;
28   }
29   FREE_VIEW(v);
30 }
```

Figure 2.12: Simplest producer consumer example.

Each process defines its own view for *int* events (lines 3 and 18). Once defined, each of the views is initialized, specifying both a type (reader or writer) and its *horizon* size. As soon as a view is initialized it is ready to connect to the respective record.

The producer process iterates 1000 times and, at each loop iteration the value of the induction variable i is assigned to the event with index i (Line 10). The *stall* primitive call guarantees that the event with index i is available and ready to be written, while *commit* specifies that the event is no longer accessed by the process, informing the record that the process no longer writes this event. The record data structure, as it is defined as connected by a single producer, immediately notifies any of its reader views.

Consumer process waits for the availability of events, through the *update* primitive. Once *update* returns, the event with index i is read and used in *printf* to display a "Hello World" message, followed by the data associated with the event i (Line 25). The *release* primitive dismisses the event with index i as it is never read by the process again.

Consumer process termination is controlled by the return value of update. While *update* returns the value provided in its index argument, the process must not terminate and iterates once more (Line 23). If *update* returns a different value, it means that the record is zombified and no new events are produced for the record.

Before terminating, both processes free their defined views through the *free_view* primitive.

2.4.1 Communication grain and peek and poke

Synchronizations primitives involve thread communication, which is an expensive operation independently of the language, runtime implementation or target architecture.

One possible optimization to such application, considering that the application synchronizes at each iteration, is to increase data communication grain size. Increasing grain size makes processes to consume and produce more data per process synchronization. Doing so, consumer processes have an increased start-up delay, since its first synchronization only unblocks when its producer has at least produced as many elements as the grain size. On the contrary, having such fine grain synchronizations increases the amount of cache incoherences, resulting in extra overhead, considering that both processes are constantly accessing the events associated with the same cache line. The benefits of grain control are mostly

dependent on the target architecture and the record events buffer implementation.

Many languages define synchronization granularity level based on the communicated data type [67, 81]. Erbium can do the same by defining a record typed with an array of *int* instead of a single *int*. However, such approach implies that neighbour connecting processes are also defined using the same data type. Moreover, such language design is not modular, as it implies transforming all but the connecting processes in order to change the communication granularity.

In Erbium, as synchronization is independent from data communication, it is possible to reduce the number of synchronizations, calling synchronization primitives less often, i.e., instead of calling the primitives for each successive index, one calls the primitive only every grain size iterations and with index increments of the same size. Such approach to communication grain size adjustment does not enforce any refactoring of the communicating processes and still benefits from the same cache and performance improvements as the other approaches.

```
1  PROCESS Producer(record int r)
2  {
3    view int v;
4    INIT_VIEW(v, WRITE, g);
5    CONNECT_REGISTERED(v, r);
6
7    for(int i = 1; i <= 1000; i += g)
8    {
9      STALL(v, i +g - 1);
10     for(j = i; j < i+g; j++)
11       v[[j]] = j;
12     COMMIT(v, i +g - 1);
13   }
14
15   FREE_VIEW(v);
16 }
```

```
1  PROCESS Producer(record int[g] r)
2  {
3    view int[g] v;
4    INIT_VIEW(v, WRITE, 1);
5    CONNECT_REGISTERED(v, r);
6
7    int k = 1;
8    for(int i = 1; i <= 1000; i += g)
9    {
10     STALL(v, k);
11     for(j = 0; j < g; j++)
12       v[[k]][j] = i +j;
13     COMMIT(v, k);
14     k++;
15   }
16   FREE_VIEW(v);
17 }
```

Figure 2.13: Synchronization granularity control of Erbium processes. Both processes are a transformation to the producer process presented in Figure 2.12, where grain communication size is adjusted to g elements. The code in Figure 2.12 is only equivalent if 1000 is a multiple of g.

Figure 2.13 are the two possible examples to control synchronization granularity. Both process transformations have the same benefits and drawbacks, i.e. produce the same overhead with respect to synchronization and data communication, and delay the communication start by grain size elements. However, the process transformation on the right breaks modularity considering the different types for its input events, forcing a refactoring to its communicating processes.

Languages like StreamIt do not need such flexibility. Its goal is to abstract the programmer from this complexities and to perform the granularity control through process fusion and blocking, adjusting granularity at compilation time as shown by Gordon et al. [34]. As Erbium is designed to be used as an intermediate language, it must have such expressiveness and close to hardware semantics in order to allow further optimizations, as ones existing in the StreamIt compiler.

Events peek and poke (direct access to events data) semantics together with its decoupled data communication and synchronization provide Erbium with sufficient expressiveness

```
1  PROCESS Averager(record float r_in, record float r_out, int size)
2  {
3    view float vr;
4    view float vw;
5    int i = 0, j = 0;
6
7    INIT_VIEW(vr, READER, size);
8    INIT_VIEW(vw, WRITER, 1);
9
10   CONNECT_REGISTERED(vr, r_in);
11   CONNECT_REGISTERED(vw, r_out);
12
13   while((UPDATE(vr, i +size)) != i +size)
14   {
15     float total = 0.0;
16
17     STALL(vw, j +1);
18
19     for(int k = 1; k <= size; k++)
20       total += vr[[i +k]];
21     vw[[j +1]] = total / size;
22
23     RELEASE(vr, i+1);
24     COMMIT(vw, j+1);
25     i++; j++;
26   }
27
28   FREE_VIEW(vr);
29   FREE_VIEW(vw);
30 }
```

Figure 2.14: Averager process — it takes the last *size* float elements from the input record (*r_in*) and computes the elements the average, writing its result to an event of its writer view (*vr*) previously connected to the record (*r_out*). It repeats the process for every new event in the input record.

to perform common streaming parallel optimizations. One example is process blocking (communication grain size adjustments) which Erbium is able to do without the need for multi-process transformations, achieving the same benefits only resorting to loop optimizations. However, these transformations should be validated through inter-process analysis. Chapter 6 further details these Erbium code analysis and the possible Erbium code optimizations.

2.4.2 Producer consumer pattern

Not all processes are either only consumers or producers. In fact, the majority of processes are consumers and producers, computing data based on its neighbour connecting processes.

Erbium with its expressiveness and modularity allows programmers to develop process libraries to simplify common tasks. Figure 2.14 is an example of such a process. It computes the average of the last *size* floats, reading values from its input record (*r_in*) and writing the result to *r_out*.

Notice the possible process re-usability, even when the programmer has no initial information of the internal process implementation. Without any knowledge of the process code, any programmer could simply use it for its own application, just like any traditional function

```
1  PROCESS Fibonacci(record int r, int n)
2  {
3    view float v;
4    int i = 0;
5
6    INIT_VIEW(vr, READER, 2);
7    INIT_VIEW(vw, WRITER, 1);
8
9    CONNECT(vr, r);
10   CONNECT_REGISTERED(vw, r);
11
12   // Initialize
13   STALL(vw, 1);
14   vw[[1]] = 0;
15   COMMIT(vw, 1);
16   STALL(vw, 2);
17   vw[[2]] = 1;
18   COMMIT(vw, 2);
19   i = 3;
20
21   while(i <= n)
22   {
23     STALL(vw, i);
24     UPDATE(vr, i-1);
25
26     vw[[i]] = vr[[i-1]] +vr[[i-2]];
27
28     COMMIT(vw, i);
29     RELEASE(vr, i-2);
30     i++;
31   }
32
33   FREE_VIEW(vr);
34   FREE_VIEW(vw);
35 }
```

Figure 2.15: Fibonacci number generator.

could.

Moreover, process modularity persists even when considering compilation, i.e., Erbium processes can be compiled individually and later used. However, in such cases, the compiler cannot specialize the process code, for example doing grain size adjustments, for the particular compiling application, but rather create multiple versions for the same process code. Erbium processes are compiled as traditional functions inheriting all of its compilation benefits and limitations. Although Erbium language was defined as an intermediate representation application, modularity is one of its very important properties and should not be neglected. Moreover, Erbium dynamic semantics allows its processes to configure views based on provided parameters. For example, the example in Figure 2.14 sets its reader view horizon size as the process *size* parameter.

2.4.3 Peek previous produced events

Many applications require processes not only to be simple consumer producers but also to access their previous produced elements, i.e. current computing data is dependent on the

past produced data. This type of parallel dependencies are very common in signal processing applications. A very well known example is H264 decoder and encoder, where previous encoded or decoded video frames and macro-blocks (square pixel regions) are dependent on the previous decoded/encoded frames and neighbour macro-blocks, as parallelized and explained by Azevedo et al. [14].

The code in Figure 2.15 is a fibonacci number generator. As anyone knows **fibonacci**, is a recursive function where:

$$fib(n) = \begin{cases} n & : \quad n \leq 1 \\ fib(n-2) + fib(n-1) & : \quad n > 1 \end{cases}$$

The process generates and commits the fibonacci sequence, stopping once the fibonacci for its parameter n is reached. In such example, the process creates both a reader and writer views for the same record. The reader view is connected as non-registered. This is possible because the process is also a registered producer of the record. Code execution ordering guarantees that the reader view is connected before the process commits any data, enforcing a deterministic initialization of the reader view. Moreover, during record initialization the reader view connection is not taken in consideration in *add_registed_views*. This property provides any application programmer with the ability to abstract itself from the internals of the process code, focusing attention in the composition of the process instances in the final application process graph.

Notice how the process performs initialization of the record for the first two fibonacci numbers from line 12 to 16.

The process continues until it reaches the computation of fibonacci of n. At each iteration the process *update*s and *stall*s for each view. As the process is also the producer for vr, *update* does not need to verify its termination, as it is guaranteed that the data should be available. Moreover, *update* will never block, however it depends on the actual runtime implementation.

In a shared memory model, one can assume that the *update* primitive usage is redundant, as both views should be using the same shared buffer. Nevertheless, such scenario cannot be guaranteed in a distributed memory architecture. In any case, if the compiler can detect that both views connect to the same record, further optimizations can be done in order to remove the redundant *update*.

Notice that although *update* is redundant in a shared memory architectures, the *release* primitive is not. In the possible scenario where the consumers execute faster then the fibonacci process, it is possible that the events in the fibonacci process are used after all the consumers released them. This is not a problem if the fibonacci process is the single producer for the record r, however, if there is another process stalling on the record r then it is possible that the record events are recycled before the fibonacci process has completed its access to it. In order to guarantee that those events are not recycled while they are still needed, it is necessary to create an extra reader view connection to the record and perform a *release* once those events are no longer necessary, as presented in the fibonacci example.

2.4.4 Broadcast pattern

As previously mentioned, the Erbium event records support multiple connections from both writer and reader views. Record data structures have shared buffer semantics, i.e., independently of its target architecture and its memory model, records are shared entities. Runtime

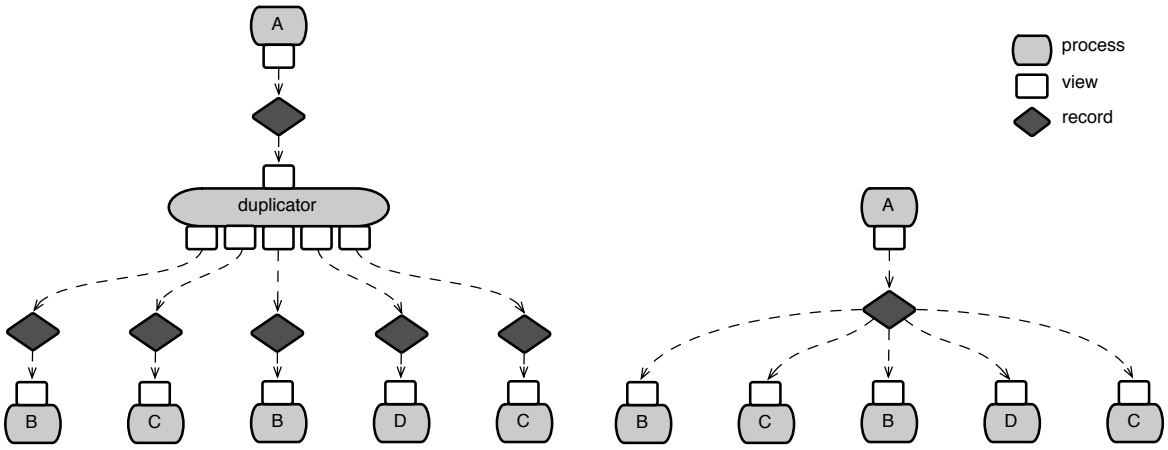


Figure 2.16: Data duplication vs. data broadcast connectivity diagrams, represented through Erbium connectivity graphs. In the Erbium connectivity graphs, rounded side rectangles represent Erbium processes and rounded edged rectangles are the views defined by the touching process rectangles. Diamond shapes are records and their edges represent writer or reader view connections.

support and compiler transformations guarantee the correct integration with different targets architecture properties.

The Erbium language and its multiple consumer and producer properties allow many concurrent processes to access, in a local fashion (views), shared centralized data within record events. Such properties, similarly, allow data distribution (broadcasts) through consumer processes without any expensive data replication. In distributed memory architectures, data is forcefully moved to the processor local memory, making broadcasts "more expensive"[1].

Figure 2.16 presents the duplication (left) and broadcasts (right) patterns represented as Erbium's network connectivity graphs. These type of graphs are widely used throughout the document, to graphically represent Erbium applications.

The left hand-side graph represents the common single producer and consumer communication approach, used by many high-level parallel streaming languages. A special process is responsible of replicating data elements from the input buffer into the many output ones. As one might expect, such approach to data-replication is both performance and memory expensive considering the required copies and memory space.

For data broadcasting, Erbium advocates programmers or code generators to use the record shared semantics. Multiple reader views should connect directly to the record, instead of being previously distributed through independent records. Erbium connections do not necessarily imply any type of data distribution through the connecting consumers. All the events available in the record are accessible by all consumers. It is the process task to decide which events are important or not for the particular usage interest.

Moreover, the same process can be instantiated multiple times, also connecting to the same record. However, unless the process instance is created with different arguments, resulting in different output data, instance replication should be avoided. One possibility, as an example,

[1] As shared memory architectures implicitly do such copies through caches, distributed memory implementations only appear inefficient but in fact can perform even better then counterpart, depending on the actual application.

is to connect multiple times the previous presented *Averager* process to the same record, while using different *size* arguments.

The `fmradio` from GNU radio package, further analyzed and benchmarked in Chapter 3, uses broadcast patterns, exploiting the application parallelism, while minimizing memory footprint and synchronization overheads.

2.4.5 Splitters and Mergers

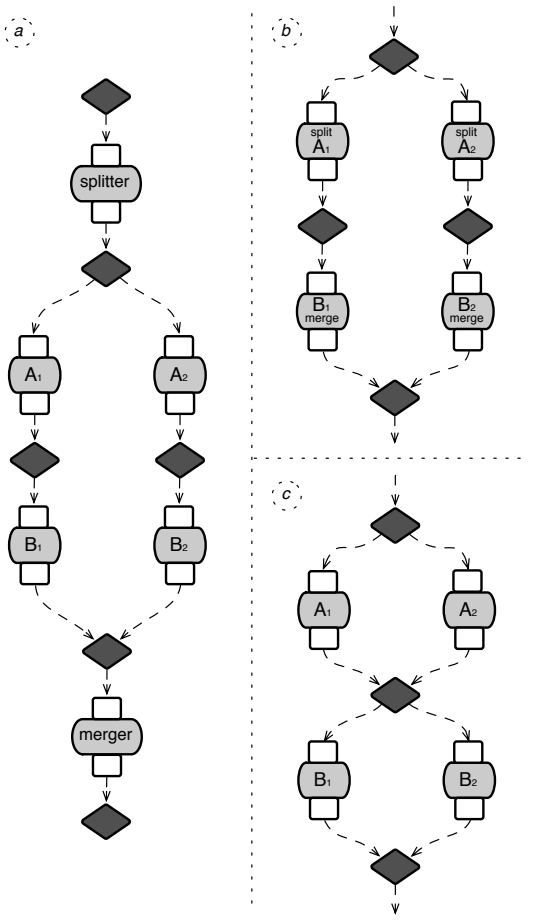
```
1  PROCESS splitter_rr (record TYPE r, nr_outs, ...)
2  {
3    view TYPE vr;
4    view TYPE vw[nr_outs]; // Dynamically allocated
5    int c = 0;
6    int size[nr_outs];
7
8    va_start(ap, nr_outs);
9    for (int k = 0; (size[k] = va_arg(ap, int)) > 0; k++)
10   {
11     INIT_VIEW(vw[k], WRITER, size);
12     CONNECT(vw[k], va_arg(ap, record TYPE);
13   }
14
15   INIT_VIEW(vr, READER, size);
16   CONNECT_REGISTERED(vr, r);
17
18   int i = 0, j[nr_outs] = { 0, 0, ... };
19   while(UPDATE(vr, i + size[c]) != i + size[c])
20   {
21     STALL(vw[c], j[c] + size[c]);
22
23     for(int k = 1; k <= size[c]; k++)   // Loop copying
24       vw[c][[j[c] +k]] = vr[[i +k]];
25
26     COMMIT(vw[c], j[c] + size[c]);
27     RELEASE(vr, i + size[c]);
28
29     i += size[c];              // Increment local indexes
30     j[c] += size[c];
31
32     c++;
33     if(c == nr_outs)
34   c = 0;
35   }
36
37   for(int k = 0; k < nr_outs; k++)
38     FREE_VIEW(vr[k]);
39   FREE_VIEW(vw);
40 }
```

```
1  PROCESS merger_rr(record TYPE r, int size, nr_ins, ...)
2  {
3    int i;
4    view TYPE vr;
5    view TYPE vw[nr_ins]; // Dynamically allocated
6    int c = 0;
7    int size[nr_ins];
8
9    va_start(ap, nr_ins);
10   for (int k = 0; (size[k] = va_arg(ap, int)) > 0; k++)
11   {
12     INIT_VIEW(vr[k], READER, size);
13     CONNECT(vr[k], va_arg(ap, record TYPE);
14   }
15
16   INIT_VIEW(vw, WRITER, size);
17   CONNECT_REGISTERED(vw, r);
18
19   i = 0; j[nr_ins] = { 0, 0, ... };
20   while(UPDATE(vr[c], j + size[c]) != j + size[c])
21   {
22     STALL(vw, i + size[c]);
23
24     for(int k = 1; k <= size[c]; k++)
25       vw[[i +k]] = vr[c][[j +k]];
26
27     COMMIT(vw, i + size[c]);
28     RELEASE(vr[c], j + size[c]);
29
30     i += size[c];
31     j[c] += size[c];
32
33     c++;
34     if (c == nr_ins)
35   c = 0;
36   }
37
38   for(int k = 0; k < nr_ins; k++)
39     FREE_VIEW(vw[k]);
40   FREE_VIEW(vr);
41 }
```

Figure 2.17: Round-robin splitter (left) and merger (right) processes.

Many high-level languages like StreamIt provide data split and merging constructs to distribute data through parallel tasks, converted later to intermediate tasks performing data splitting and merging. As mentioned in the previous broadcast use case, using extra tasks/processes for splitting and merging is not the best when considering Erbium. In any case, Erbium expressiveness is not restricted to any specific implementation.

The code in Figure 2.17 present round-robin data partitioning splitter and merger processes. Data distribution is defined based on the provided arguments (variable number of arguments). The *nr_outs* and *nr_ins* parameters define the number of actual records in which either the splitter or the merger will distribute or combine data. The remaining parameters are pairs of both the size of events, each of the records should contribute for the actual split or merge, as well as the record in which those events are written or read, respectively. The statement `run splitter_rr(in, 2, out1, 5, out2, 10)` splits the content of the record *in* through the 2 (second argument) records *out*1 and *out*2, copying the content of the first 5 events to *out*1 and the next 10 events to *out*2, repeating its execution until the




```
1  PROCESS Averager(record float r_in, record float r_out,
2                   int average_size,
3                   int payload_size, int offset, int size)
4  {
5    view float vr;
6    view float vw;
7    int i = 0;
8
9    INIT_VIEW(vr, READER, av_size);
10   INIT_VIEW(vw, WRITER, 1);
11
12   CONNECT_REGISTERED(vr, r_in);
13   CONNECT_REGISTERED(vw, r_out);
14
15   int first_index = offset;
16   int last_index = first_index + size;
17   i = first_index;
18   RELEASE(vr, i);
19   COMMIT(vw, i);
20
21   while((UPDATE(vr, i + av_size)) != i + av_size)
22   {
23     float total = 0.0;
24
25     STALL(vw, i + 1);
26
27     for(int j = 1; j <= av_size; j++)
28       total += vr[[i +j]];
29     vw[[i +1]] = total / av_size;
30
31     RELEASE(vr, i+1);
32     COMMIT(vw, i+1);
33     i++;
34
35     if(i == last_index)
36     {
37       first_index += payload_size;
38       last_index = first_index + size;
39       i = first_index;
40       RELEASE(vr, i);
41       COMMIT(vw, i);
42     }
43   }
44
45   FREE_VIEW(vr);
46   FREE_VIEW(vw);
47 }
```

Figure 2.18: Three possible split and merge scenarios for the same set of processes instances (left side) and an *Averager* process code with embedded split and merge logic (right side).

record *in* terminates. The left most graph in Figure 2.18 is an example connectivity graph for split and merge processes.

Erbium does not advocates/needs splitter or merger processes. Such approach not only requires an extra thread and more resources to allocate the needed extra records. Although providing Erbium with a modular approach to data splitting and merging, it also limits Erbium processes expressiveness. For example, using the splitter process, worker processes are limited to access to the splitter predetermined data. Splitter (and merger) processes imply fully data parallel workers, i.e., workers must only depend on subsets of the input and no events can be shared between the different worker processes. For example, data parallelizing the previous presented *Averager* process (Figure 2.14) is not possible using the presented splitter and merger.

Data distribution can be done within the target process code by providing the process instance with enough information to consume and produce only a subset of the events. In most cases, converting processes into data parallel versions only requires a small code adaptation. Figure 2.18 presents a data parallel *Averager* implementation, as well as the Erbium possible data partitioning schemes (three connectivity graphs).

The presented *Averager* algorithm contains three new parameters allowing to define which

range of the input events is consumed and produced by a particular process instantiation. The majority of the process code is left untouched, only requiring the inclusion of an index shifting to the instance relevant indexes at each iteration (Lines 15-17 and 37-39). Moreover, each time a shift of the indexes is computed, the non-relevant events are released and committed, in conformity with what was previously said in the example of Figure 2.8.

None of the graphically presented approaches is preferable over the other. Each one has its own benefits.

(a) The splitter and merger independent process implementation allows for a more modular implementation, allowing programmers to compose applications with producer and consumer processes,
(b) Embedding the process split and merge functionality in first and last processes of the data partitioning provides the flexibility of explicit preform data splitting and merging. In any case, such approach reduces modularity considering the required process specialization. Nevertheless, none of the processes used between the splitter and merger processes needs to be changed. Performance improves thanks to the reduced number of process instances required,
(c) The last approach takes full use of multiple producer consumer records, reducing the number of records from the previous presented approaches. The *Averager* code, as presented in the figure, is implemented in this way.

Like previously said, none of the presented approaches is preferable over the other. This is the case because, in order to decide for a particular approach, it is necessary to also define an application, memory model, thread scheduling policies and even to analyze the runtime support implementation. More importantly, Erbium supports the expressiveness to exploit all three work splitting (data distribution) approaches (left side of Figure 2.18), eventually supporting a greater number of high-level languages through its intermediate representation.

2.4.6 Process hand-over

Many applications might require Erbium to have a more adaptive behaviour with respect to external or even internal process factors. Examples are event driven algorithms that require to adapt or change behaviour during execution. One can claim, correctly, that a single process could implement all of the possible cases and change state in order to adapt. However, if the adaptation involves disconnecting part of the process network while still sharing the same data input record, such extreme reconfiguration would be impossible.

Process hand-over enables such dynamic network adaptability by allowing processes to disconnect and give the opportunity to other processes to connect in their place.

Figure 2.19 is a simple example of a process hand-over. Both processes apart from the usual record parameters also include two view connection identifiers *view_id*, obtained externally to the process using the primitive *get_new_view_id*, as previously explained.

At every process iteration both processes verify if the process should perform an hand-over based on the output of the function *should_switch_algorithm*. This function takes its decision solely based on the current process state, i.e., its own reader view events data, meaning the process code should never access global variables or other process external data structures.

When a process requires to hand-over, it frees its views specifying a process hand-over is about to occur (second argument of *free_view*). Before terminating, the process creates an instance of the substituting process.

```
1  process Algorithm1(record Data r1, view_id id_r1,
2          record Data r2, view_id id_r2)
3  {
4    view int vr;
5    view int vw;
6    init_view(vr, READER, 1);
7    init_view(vw, WRITER, 1);
8
9    connect_registered(vr, r1, id_r1);
10   connect_registered(vw, r2, id_r2);
11
12   int i = view_tail(v) + 1;
13   while(update(vr, i) == i)
14   {
15     hand_over = should_switch_algorithm(vr);
16     if(hand_over)
17       break;
18
19     stall(vw, i)
20
21     ... /* algorithm 1 */
22
23     commit(vw, i);
24     release(vr, i);
25     i++;
26   }
27
28   free_view(vr, !hand_over);
29   free_view(vw, !hand_over);
30
31   if(hand_over)
32     run Algorithm2(r, id_r1, rw, id_r2);
33 }
```

```
1  process Algorithm2(record Data r1, view_id id_r1
2          record Data r2, view_id id_r2)
3  {
4    view int vr, vw;
5    init_view(vr, READER, 1);
6    init_view(vw, WRITER, 1);
7
8    connect_registered(vr, r1, id_r1);
9    connect_registered(vw, r2, id_r2);
10
11   int i = view_tail(v) + 1;
12   while(update(vr, i) == i)
13   {
14     hand_over = should_switch_algorithm(vr);
15     if(hand_over)
16       break;
17
18     stall(vw, i)
19
20     ... /* algorithm 2 */
21
22     commit(vw, i);
23     release(vr, i);
24     i++;
25   }
26
27   free_view(vr, !hand_over);
28   free_view(vw, !hand_over);
29
30   if(hand_over)
31     run Algorithm1(r, id_r1, rw, id_r2);
32 }
```

Figure 2.19: Simple example of process hand-over. *view_tail* is a pseudo function to retrieve the index left by the disconnecting process.

Hand-over process instantiation should only occur once the view connection identifiers are already freed from views, otherwise both processes will access the record events concurrently, resulting in a non deterministic data store in the shared record events.

Complex hand-over

Like mentioned before, with process hand-over, it is possible to dynamically redesign the network of processes. Performing an hand-over of such level implies to guarantee that every process in the network deterministically connects and disconnects from the application process network. It is not the job of the language to guarantee the determinism in this case, but rather the programmer responsibility, respecting the language rules.

In a multi-process hand-over, one must perform the hand-over of the graph boundary processes, implying the synchronization between the head and tail process of the substituting network.

Figure 2.20 is a graphical representation of an hypothetical network of processes. The figure is split in the intermediate stages of a process network hand-over. The numbered edges represent the order of the required operations occurring during the hand-over. The left most diagram is the initial state of the network. The dashed rectangles represent the boundaries of the substituting network of processes.

In this example the process A is responsible to detect when an hand-over is requested. When it happens, the process commits any remaining events that should be computed within the existing network of processes. Once no more data need to be produced, A commits a message to the record shared with C (rs) announcing an upcoming hand-over (1). At this point, A frees any reader view for an hand-over, keeping the connection identifier open and frees its writer view permanently (2).

The process A instantiates all the processes of the substituting network (3), creating

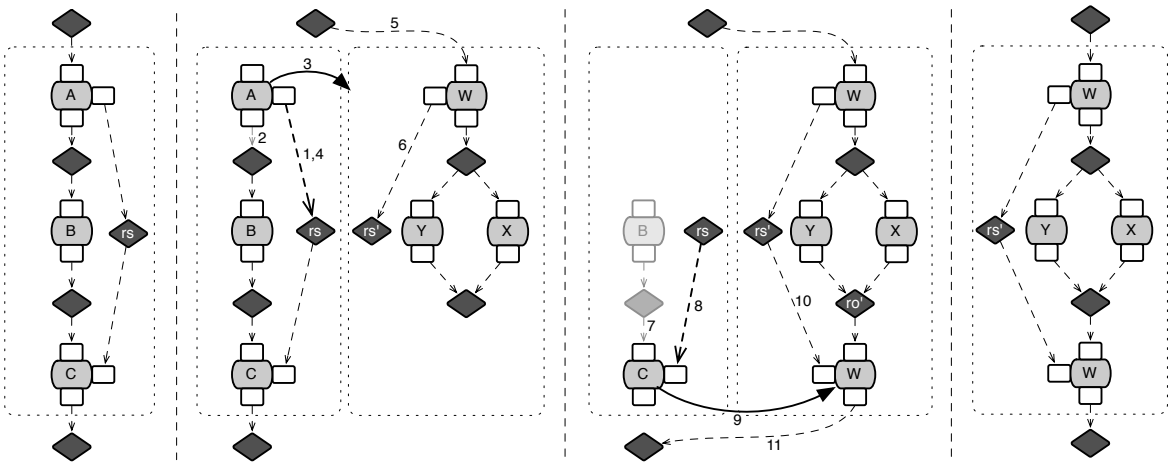


Figure 2.20: Complex hand-over of a network of processes.

all the required internal records and composing the network by providing those records as arguments of the implied processes instantiations.

As a last step, A commits again to the record shared with C, this time providing references to the new records created during the new network instantiation (4), followed by the freeing of the respective view (rs).

At this point, the process W from the new network takes the place of A in the application and performs the connection to the view identifier freed by A (5). At this point any record data available in the top most record is consumed by process W and the new network immediately starts its work. This implementation allows a continuous application progress by quickly initiating the new network of processes.

As the process A writer views are freed permanently, zombification is propagated through its network stream of processes. Once zombification reaches the process C (7), as the process has no more data to consume, it verifies if the record rs contains any "announcement", meaning an hand-over (8). If this is the case it should also contain an extra commit with the new records created through the network instantiation at step 3.

At this point and depending on the actual announce at record rs the writer views are freed to hand-over, The process C instantiates the tail process missing of the new network. It does it, providing the new process with both the view connection identifier (of the previous freed writer views) and the previous reference record (provided by process A in step 4)

After instantiation, the process W connects to all the necessary records, more precisely at rs' which is substituting the previous rs record (10) and connecting to the network output record using the view identifier (11).

After instantiating process W, the process C terminates and the network hand-over has completed.

Such complex process hand-overs are not intended to be coded by hand, but instead to be generated when compiling more abstracted and simplified high-level languages.

2.4.7 Data locality/pre-fetching

The Erbium language also supports data locality optimizing primitives. This is the case of the *transfer* primitive allowing processes to anticipate which events are soon read or written

```
1  PROCESS Producer(record int r)
2  {
3    view int v;
4    INIT_VIEW(v, WRITER, 1024);
5    CONNECT_REGISTERED(v, r);
6
7    for(int i = 1; i <= N; i += 128)
8    {
9      TRANSFER(v, i+127)
10     STALL(v, i+127);
11     /* Produce events v[[i..i+127]] */
12     COMMIT(v, i+127);
13   }
14
15   FREE_VIEW(v);
16 }
```

```
16 PROCESS Consumer(record int r)
17 {
18   view int v;
19   INIT_VIEW(v, READER, 1024);
20   CONNECT_REGISTERED(v, r);
21
22   int i = 0;
23   TRANSFER(v, 1024)
24   while(UPDATE(v, i+128) == i+128)
25   {
26     /* Consume events v[[i+1..i+128]] */
27     RELEASE(v, i+128);
28     TRANSFER(v, 1024 +i +128)
29     i += 128;
30   }
31   FREE_VIEW(v);
32 }
```

Figure 2.21: Transfer primitive example.

by the process.

Both runtime and compilers should use this primitive as a hint to temporal and spatial locality of the communicating data. The primitive is independent of the architecture memory type, being it either a cache pre-fetch in case of a shared memory architecture or an explicit asynchronous memory transfer call in distributed memory architectures.

This primitive does not immediately triggers a transfer operation. Instead, and depending on the target architecture and runtime implementation, it would store the ranges of events provided in its arguments, which would later be used by the synchronization primitives calls (*update* and *commit*) to initiate the data transfer. Moreover, no *transfer* primitive by itself is responsible for any explicit data transfer, but only the combination of the *transfer* primitive and the synchronization primitive calls. As an example, if a producer commits for a range already already expected by a consumer transfer call, the data transfer is initiated at the commit primitive call. Although the transfer primitive is executed before the commit, in the consumer process, the transfer only initiates when the events are committed. Nevertheless, the transfer decision is tightly related to the target architecture and runtime implementation. As an example, in distributed memory architectures, the transfer can only occur when both producer and consumer are ready. The producer must have committed the data, but also the consumer must have sufficient resources in its local memory to accommodate the transfer waiting events.

Figure 2.21 is an example of a producer consumer application using transfer primitives to anticipate data communication between processes memory regions.

On the producer side, transfer primitives are executed at any point before the *commit* primitive. As there is only a single producer, *transfer* is used for all the events committed.

The consumer side example is more interesting considering *transfer* always requests to fill the view sliding window.

In a multiple producer and consumer example, the *transfer* primitive has a much greater interest and impact. In distributed memory architectures, the transfer primitive also allows to disambiguate which producers are responsible for the respective event index ranges, being a requirement to Erbium's determinism. Without the transfer primitive it would be impossible to merge the multiple buffers associated with the concurrent producers. Moreover, the

transfer primitive allows to directly communicate data between the producer and consumer processes without ever merging all the distributed buffers into a centralized one, reducing the necessary number of copies and its expensive star communication pattern.

The Chapter 3 further analyzes the *transfer* primitive implementation for both shared and distributed memory architectures.

2.5 Summary

High level languages and architectures obey very different design goals and constraints. Compilers bridge the gap between these abstractions, by adapting and optimizing languages into closer-to-hardware representations. This is well understood for sequential languages. But parallel languages are yet to find a core support from compilers, mainly because no representation has sufficed to all the variety of architectures and parallel languages. Streaming and data-flow languages expose parallelism by partitioning the application into concurrent tasks connected through dependent data or data streams. We focus on this very expressive form of parallelism because of its intrinsic semantical properties (determinism in particular), and because it exposes more static properties for compilers to transform the code and adapt it to the parallel target.

This chapter presented Erbium, a streaming language capable of representing higher level streaming languages as an intermediate representation or being used as a performance language for efficiency programmers. We detailed the semantics and syntax of its main data structures (records and views), and explain some particular use cases for the language, such as data peek and poke with decoupled synchronization, multiple producer and consumer record accesses, data broadcasting, work-splitting and process hand-over in an attempt to demonstrate all of the Erbium language properties and features.

Chapter 3 further details the language, by explaining a x86 shared memory implementation of the language data structures and primitives exploiting the architectures properties. It compares with other types of implementations and presents a possible distributed memory implementation. Later chapters explain how to embed Erbium as a compiler intermediate representation, how to lower higher level languages into the intermediate representation and how to optimize it.

Chapter 3

Runtime

Different processors with the same instruction set architecture can contain many subtleties generating very distinctive execution times from the same executing binary. Compilers must be precisely configured for each precise machine, sometimes involving specific optimization passes or pass selection. On multiprocessor architectures, these subtleties include the number of available cores, the structure of the cache hierarchy, the nature of the interconnect and the coherence algorithms. Such variations involve precise customization of both compilers and runtime libraries. In addition, the runtime libraries themselves should be specifically compiled for the target micro-architectures and interconnect.

The first Erbium implementation relied solely on a runtime library and macro expansions. This implementation turned out to be too slow, considering how high-level such library and set of macros were. To obtain an efficient implementation, it quickly appeared that only part of the language should have been implemented through the runtime library, where most commonly used language constructs were directly expanded by the compiler. The most obvious case was the translation of view/record accesses into low-level buffer/memory accesses: function calls, the monotonic indexing buffer addressing and control flow overheads must be systematically eliminated considering the number of times such abstractions are used. On the other hand, synchronization constructs cannot be lowered and optimized by the compiler as easily as memory accesses. While optimizations for buffer accesses are relatively portable and result in compact code, at least for shared memory architectures, this is not the case with Erbium synchronization. Synchronization is dependent on many details of the target architecture, memory model and/or communication API. As a result, optimized synchronization code can be quite large and complex. Hence the need for runtime support for synchronization in general. Moreover, synchronization, record and view allocation, initialization, termination and process instantiation are implemented within a Erbium runtime library, abstracting compilers from the complexity of such operations.

This chapter describes the runtime library for Erbium, called *libEr*, and its implementation. The described implementation supports shared memory x86 instruction set architectures. Its memory consistency model allows for a busy-waiting, lock-free implementation not relying on memory fences or atomic operations. Such implementation makes more efficient in applications where the number of processes is less than or equal to the number of CPUs, considering the lack of a good legacy scheduling policy for such type of concurrent applications.

In addition to the busy-waiting solution, this chapter also describes alternative designs

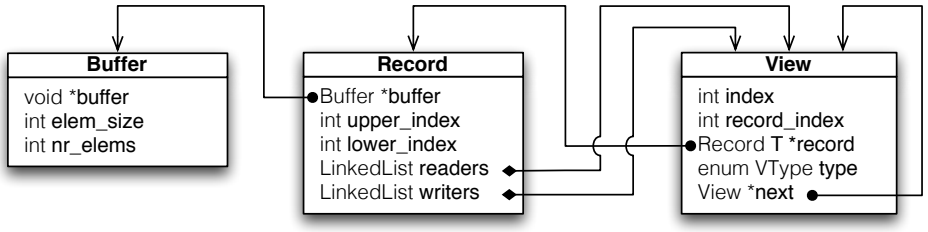


Figure 3.1: Simplified libEr record and view data structures.

supporting lazy-waiting and user-level threading, providing support for more power-efficient systems. These designs also consider bigger, over partitioned, and load unbalanced parallel applications, as well as alternative operation environments, including Non-Uniform Memory Access (NUMA) architectures.

Fastflow is closely related to the libEr [9]. It is a C++ programming pattern for streaming applications dedicated to shared-memory platforms. Its lock-free, fence-free synchronization layer has comparable performance to our single-producer single-consumer record. It supports multi-producer multi-consumer communication at the expense of an additional arbitration thread and memory copying. Because index-range negotiation is exposed to the compiler in the intermediate representation, our runtime achieves lock-free, fence-free multi-producer multi-consumer communication without these overheads.

The chapter ends with the concept design of a distributed memory implementation, to illustrate the portability potential of Erbium, and how Erbium and libEr is able to support explicit communications in particular.

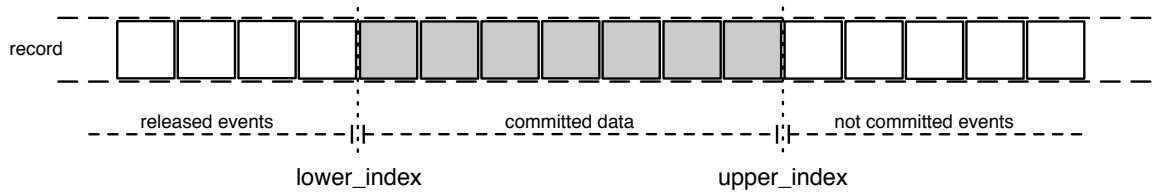
3.1 libEr implementation

libEr is the actual implementation of Erbium runtime system, implementing Erbium support for process creation, synchronization and its data structures (record, view and buffers) allocation, initialization and termination. The presented version of libEr supports the commonly known X86 instruction set architectures, exploiting its shared memory model, its cache coherence and memory consistency models, minimizing synchronization cost and primitives overhead.

Similarly to the language definition, the runtime library also defines *record* and *view* data structures. These data structures are not directly associated with any specific event type, as presented in the previous chapter record and view definitions. Instead the record is connected to a buffer data structure that is allocated using the type declared in the language record and view definitions.

Figure 3.1 diagram shows the content of both the record and view data structures. Each record is associated with a buffer data structure, responsible to store all of the record events data. The buffer is composed of a non-typed memory region, size information of the original type and buffer capacity.

The record data structure includes two lists, used to keep track of all the connected views for reader or writer views. One of the lists stores pointers to the writer views while the other pointers to reader views. In the diagram, the lists are represented as "*LinkedList*", which is just an abstraction to all the necessary pointers to the *view* data structure necessary in



$$upper\_index = MIN\{i : \forall view \in record.writers, i = view.index\}$$
$$lower\_index = MIN\{i : \forall view \in record.readers, i = view.index\}$$

Figure 3.2: Record upper/lower delimiters and its computation.

an efficient linked list implementation, more precisely pointers to the first and last linked list view elements.

Upper and lower indexes are the attributes used within the execution of the *update* and *stall* primitives, respectively. The upper index separates the events (buffer positions) still being written by any of the connected views from the already *committed* and *ready* events. The lower index on the other side is the index separating events still being accessed (read) by any view, from the no longer necessary and released events. Please refer to Chapter 2 to remember event states and Erbium synchronization.

The view data structure in its simplest form defines only an index and a pointer to the record it is connecting to. Its index attribute is incremented by *commit* or *release* primitive, depending if the view is a writer or a reader, respectively. In any case, its value should be the maximum index ever provided through these primitives. In other words, the index attribute is the event index marking the tail of the view sliding window.

Within its simplest form view data structure is inefficient and is not recommended. Two more attributes are necessary for the data structure. The *type* attribute classifies the view as either a *reader* or *writer*. This attribute avoids the record to always traverse the two linked lists in order to identify the view type. The *record_index* attribute is a "caching" attribute, meaning that, during the synchronization *update* or *stall* calls, the value from *upper* or *lower* indexes is copied to *record index* attribute, reducing the number of accesses to the shared record data structure and eventually reducing synchronization overhead of future similar synchronization primitive calls.

Index based synchronization

Process synchronization occurs as an interaction between the views and the record data structures using the view synchronization primitives. Progress is acknowledged to the runtime system through *commit* and *release* primitives, where the local view *index* attribute is written, changing the tail of the view sliding window, specifying the highest no longer accessed index position (Algorithms 1 and 2).

The *update* and *stall* primitives block process execution while not enough record events are in "ready" or in "no longer needed" state, respectively.

The *ready* or *no longer needed* events state boundaries are flagged by *upper* or *lower* indexes, also known as *record indexes*. The upper index is computed based on the arithmetical minimum of the index of all the record connected writer views. Lower index is computed similarly to upper index but instead based on the index of the reader views. Please consider

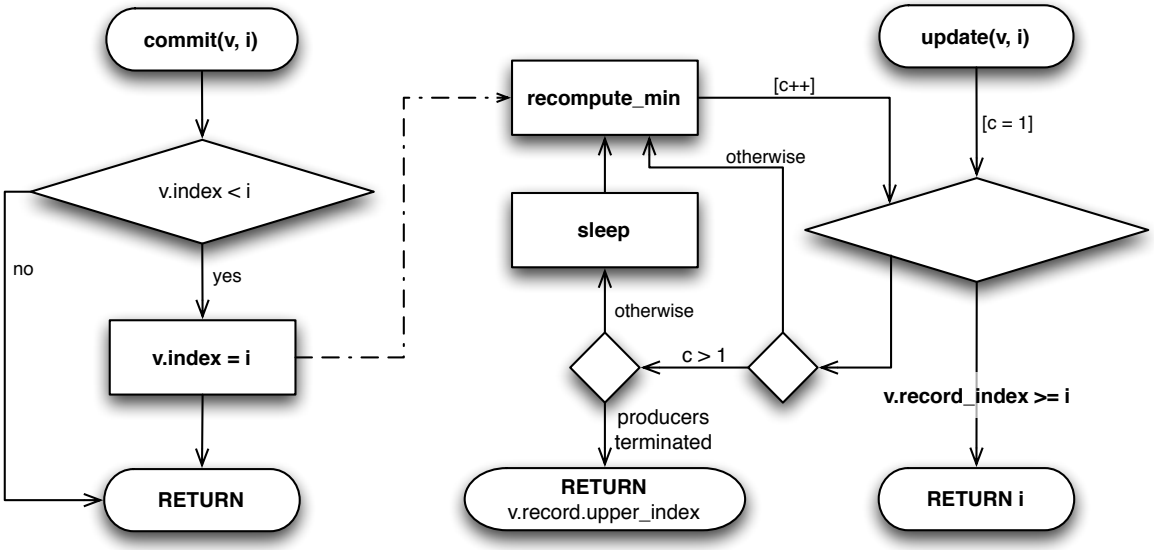


Figure 3.3: Minimalist *update* and *commit* primitives implementation state diagrams.

Figure 3.2 that outlines a diagram showing a record and its respective *record_indexes*. Below are the *upper* and *lower* indexes calculation equations based on the connected views.

All the connected writer views have an index above its record *upper* index. Reader views always contain an index between the *lower* and *upper* index.

The *stall* or *update* primitive calls block execution if the requested index, provided as argument, is bigger than lower or upper index, respectively. Primitive semantics were explained in Chapter 2.

Figure 3.3 shows a shared memory implementation, as a state diagram, of both *commit* and *update* primitives. The *commit* primitive writes its local view *index* attribute if i is bigger than the current view index, validating for the monotonicity of the view index. The *update* on the other hand verifies if *record_index* (its local cached *upper_index* attribute) of the view is bigger than its argument i in which case it returns i immediately. Otherwise, it checks if *upper_index* is different from *record_index* in which case it copies its value to *record_index* view attribute and rechecks the initial condition. If the condition is still false and the *upper_index* is equal to *record_index*, the record *upper_index* attribute is computed as expressed in Figure 3.2 (also known as minimum computation), copying its result to the *record_index* view attribute. Once a minimum is computed, it loops again and verifies if the unblocking condition has succeeded. If after the upper index minimum computation the condition is still false, it verifies if the producer processes have been terminated, in which case it should return the value from upper index. Within the following loop iterations, before the minimum computation execution, the process will "sleep"[1] in which case the thread scheduler might context switch current thread in an attempt to execute other waiting processes (Algorithm 3). Termination detection (record zombification) is explained later in the chapter.

[1]The sleep operation used in both *update* and *stall* is implemented using an x86 assembly instruction that performs a micro second sleep. The sleep results in a thread context switch in case there are other threads waiting to execute. This sleep is non blocking, i.e., no thread needs to wake it.

Algorithm 1 Writer view *commit*

```
1: function COMMIT(view,index)
2:     if view.index < index then
3:         view.index ← index
4:     end if
5: end function
```

Algorithm 2 Reader view *release*

```
1: function RELEASE(view,index)
2:     if view.index < index then
3:         view.index ← index
4:     end if
5: end function
```

The *stall* and *release* have similar interaction with record and view data structures. However, *stall* predicts if it should block by comparing argument index i against the *lower_index* record attribute. *Stall* does not return any value and does not check for record termination.

Contrarily to *update*/*commit*, *stall*/*release* have semantical differences with respected to their index arguments. The *stall* index refers future newly available events while in *release* such index refers to no longer needed ones. In other words, the release of an event with index i does not synchronize with a stall executed for the same index i, but for the index $i + size$, being *size* the number of events capable to be stored in the record. It is clear that such indexes, although related must be shifted before being compared. The shift is the number of events capable to be stored in the record (*record.buffer.nr_elems*), i.e., the record buffer size. Instead of performing such shift at every call to *stall* primitive, the shift occurs each time *record_index* is assigned (lines 8 and 10 of Algorithm 4). This shifting and the termination are the only implementation differences between *update* and *stall* primitives.

As a reminder from previous chapter, *stall* unblocks whenever enough events have been *released* by the consumer processes, or in other words when the record buffer contains enough capacity to allocate the newly requested events through *stall*.

Algorithm 3 Update

```
1: function UPDATE(view, index)
2:     iteration ← 1
3:     while view.record_index < index do
4:         if iteration > 1 then
5:             if is_record_zombie(view.record) then
6:                 return view.record_index
7:             end if
8:             sleep()
9:         end if
10:        if view.record.upper_index ≥ index then
11:            view.record_index ← view.record.upper_index
12:        else
13:            view.record_index ← compute_min(view.record, Writers)
14:        end if
15:        iteration ← iteration + 1
16:    end while
17:    return index
18: end function
```

Algorithm 4 Stall

```
1: function STALL(view, index)
2:     iteration ← 1
3:     while view.record_index < index do
4:         if iteration > 1 then
5:             sleep()
6:         end if
7:         if view.record.lower_index + view.record.buffer.nr_elems ≥ index then
8:             view.record_index ← view.record.lower_index + view.record.buffer.nr_elems
9:         else
10:            view.record_index ← compute_min(view.record, Readers)
11:                                +view.record.buffer.nr_elems
12:        end if
13:        iteration ← iteration + 1
14:    end while
15:    return index
16: end function
```

Record indexes computation

Consider a consumer example performing an update. The consumer itself does not know how many reader or writer views the record is connected by. It performs update to request the events up to a specific index. *Update* as seen in Algorithm 3 starts by verifying if its local *record_index* attribute (cached *upper_index*) is bigger than the requested argument index (Line 3). When it fails a minimum computation of all the writing views executes, calculating a new *upper_index* attribute value (Line 13).

The minimum computation takes the view arguments (*record* and *type*) and decides which of the view linked lists and record indexes it should compute the minimum for. Computing the minimum occurs through a linked list traversal comparing the index of all its views. Algorithm 5 presents the minimum computation for both the *record indexes*, depending on the *type* parameter.

Algorithm 5 Minimum record view index computation, used in stall and update

```
1: function COMPUTE_MIN(record, type)
2:     if type = Writers then
3:         list ← record.writers
4:         index_ptr ← &record.upper_index
5:     else
6:         list ← record.readers
7:         index_ptr ← &record.lower_index
8:     end if
9:     tmp ← ∞
10:    for view ∈ list do
11:        tmp ← MIN(tmp, view.index)
12:    end for
13:    *index_ptr ← tmp
14:    Return tmp
15: end function
```

Considering Erbium's multiple producer and consumer support, it can occur that dif-

ferent *stall* and *update* calls are executed simultaneously resulting in concurrent minimum computations. The concurrent executions of minimum computation are acceptable and valid for the presented implementation, as is explained later in the chapter.

Events data access

Erbium's events data is stored in previous presented buffer data structure. Erbium does not enforce any particular buffer implementation. However, compatible buffers must efficiently perform the following operations:

- provide a non conflicting memory position for each of the record events based on its monotonic index,
- know and provide its current maximum capacity,
- when resizable, the buffer should provide index based function implementations to request more resources as well as free no longer needed resources.

The presented libEr implementation uses an unmanaged[1]constant sized circular buffer. Each buffer position is associated with a particular single event through its "life cycle". Once an event is recycled, the specific buffer position is reused for a new event. Monotonic index events data position is computed using *modulo* (%) operator with the buffer size.

$$view[[i]] \Rightarrow buffer[\, i \ \% \ buffer\_size \,]$$

This operation eliminates buffer overflows by reassigning same positions to indexes bigger than the actual buffer size.

The *stall* primitive blocks process execution as a consequence of resources unavailability. In other words, no free buffer positions are available. In order for the *stall* primitive to identify available resources, the buffer should, as said before, be able to provide its current capacity, meaning, the maximum number of elements storable in the buffer. In case of a dynamic buffer implementation, before the capacity is queried, the buffer resources request function is called passing the stall index value as argument. If not enough buffer positions are allocated, such call can allocate more memory before, allowing the buffer to have enough resources for the current *stall* call.

Algorithm 4 (line 8) accesses the buffer capacity to detect if enough resources are available for current *stall* call.

Figure 3.1 shows a simplified buffer data structure. Considering its constant allocation size, and its typeless memory allocation, this data structure does not require any runtime implementation apart from its data structure allocation. Other types of buffer implementations might require functions to dynamically determine buffer capacity and perhaps perform buffer accesses. Runtime computed buffer accesses are not recommended considering the frequent execution for such operations.

The Erbium language and its runtime library do not support any buffers accesses abstraction. Compilers should perform its lowering directly to the specific operations depending on the type of the buffer implementation. Consider the following lowering operation ($\Rightarrow$) performed by our current compiler implementation, where $view[[i]]$ is a request to access data

[1]No verification is done to control usage of the buffer. Erbium synchronization when used appropriately will avoid buffer overflows or reading non written elements.

at index i from *view* and *buffer* is the view related *buffer* data structure, more precisely, $buffer \leftarrow view.record.buffer$ as can be perceived from previous presented Figure 3.1.

$$view[[i]] \Rightarrow *(buffer.data + (buffer.elem\_size \times (i \ \% \ buffer.nr\_elems)))$$

This code generation, apart from the final buffer index position computation (based on *modulo* operation) is very similar to what a mainstream compiler generates for an array access. Benefits of such lowering are later explained through Chapter 4.

This specific implementation does not imply dynamic buffer allocation or deallocation. If a precise buffer implementation requires to know which events indexes (monotonic buffer indexes) are being synchronized, in order to allocate or deallocate buffer elements, this must be implemented inside the *stall* implementation (for allocation) and *release* (for deallocation).

3.1.1 Implementation discussion

libEr exploits X86 architecture properties to the maximum. One example is its synchronization primitives, performing its tasks through shared memory variables (indexes).

How is this possible? Why does it work?

Like previous mentioned Erbium language synchronization is based on monotonic index variables. Each independent process contains its private index (in a view), to which it announces progress by incrementing this index through the usage of *commit* or *release* primitives. *Write* access to this index variables is only performed through the usage of those primitives and only from the owning process (thread), leaving no chance for concurrent racing writes.

Update and *Stall* primitives using the record data structure traverse all the views, *reading* the current perceived view index and computing the minimum index. Such minimum computation allows to identify current progress of all combined reader or writer views in the system.

Although different view indexes are not written atomically, being so perceived at different times through the system cores, such inconsistency is not problematic considering the indexes are monotonic, and impacting all of the previous index value events.

Nevertheless, monotonicity of the indexes does not guarantee execution ordering. Furthermore, one of the most relevant x86 properties is its memory consistency model, more precisely total store ordering (TSO) [43, 61, 74]. TSO guarantees that all stores are perceived in the same order as the operations are executed by the particular cores, i.e. two store operations executed by a single core are never perceived out of order. Moreover, load operations always read the values of the latest stored operations.

Although TSO is irrelevant, in respect to the internal implementation of synchronizations, it is essential when considering other memory operations that depend on the synchronizations. Like so, TSO consistency model certifies that any memory operation, in the record buffers is perceived in any other CPU before, for example, the respective commit is perceived, guaranteeing the monotonicity of the record and view index values.

Non-TSO architectures require memory barriers at *commit* and *release* guaranteeing buffer writes are perceived through all the system cores before the *commit* or *release*. Some architectures might include some less expensive memory consistency operations guaranteeing TSO.

What about index operations. Shouldn't those be atomic?

Although a view is only accessed by its defining/owning process, and only either *commit* or *release* write into the view index, it is possible that the view index variable is incorrectly perceived.

Let's assume that we typed all the indexes in libEr as 64-bit integers. Compiling libEr code on a 32-bit architecture, the compiler will turn any single index write into at least two write instructions, one writing the most significant bits and the other the less significant ones.

Such implementation introduces a coherence problem considering the apparent short period which the index is invalidated. Since this write operation is performed without enforcing atomicity, any of the executing CPUs can eventually read an invalid index value. Moreover, the apparent short period can be "amplified" if we consider that the process thread can be switched (stalled) in between the execution of these two instructions.

To guarantee atomicity of every index writes, one can perform write operations using atomic operations. Nevertheless, such operations introduce extra overhead when comparing with non protected memory stores.

The approach used in the libEr implementation is to type all indexes variables with a single instruction load and store capable type (32-bit integer in the previous example). Moreover, the compiler code generation should guarantee to always generate a single instruction from all its index operations.

As cache coherence algorithms execute at the instruction level, a single instruction operation is never half-perceived by other thread, and no threading system can context switch at the middle of an executing instruction.

Considering that all the index variables in the runtime are precisely typed to allow single instruction write, the index values are never perceived as invalid by concurrent threads (CPUs).

What about record upper and lower indexes?

Record upper and lower indexes also have the same behaviour. Nevertheless, in the presented implementation, those indexes are written by multiple threads concurrently. Such concurrent writes are acceptable considering that all the writes are the result of the same computation (minimum computation), even when perceiving different values. Which means that all the writes are the result of the same traversal. If the results are different in two concurrent executions, the result is simply a coherence delay and eventually every concurrent thread returns the same result. Furthermore, minimum computation results are always monotonic in the context of a single core, as are all its record connected view indexes. In other words, in the context of a single running process, record monotonicity is never broken.

What happens when processes/threads perceive outdated view indexes values during minimum computation?

When the update or stall primitives are executed, unless the current upper or lower indexes, respectively, are big enough, all the view indexes are traversed and an arithmetical minimum is computed (minimum computation). If any of the views indexes are not updated for the specific thread CPU local cache, a different minimum is computed based on the current visible index. As there is a single writer to each of the views indexes and thanks to its monotonicity, the perceived values are also visible as monotonic. If a new minimum is high enough to satisfy the *update* or *stall* condition, nothing is done and the process execution

proceeds, otherwise the primitive loops until a high enough minimum is reached as previous explained.

What about concurrent executions of compute_min. Can those be a problem?

Concurrent computation of minimum can represent a problem considering that slower processes might in fact break monotonicity of the upper and lower record indexes, by writing over a higher record index value. In any case, breaking monotonicity is not a problem as long as any of the connected processes is able to correct the index, either by detection and avoidance or by repeating its computation. As the current implementation spinlocks executing *compute_min* until the primitive condition is satisfied, such problem never arises.

More lazy implementations require mutual exclusivity during *compute_min* execution, as well as signaling mechanisms waking up any lazy (waiting) processes.

3.1.2 Process instantiation and termination

The process creation is implemented in libEr using existing support thread libraries. Threading libraries are tightly related with target operating system and greatly depend on machine architecture properties and extensions.

Runtime support for Erbium processes creation and termination is defined as an abstraction layer for code generation. This abstraction allows compilers to easily support Erbium processes without directly lower code to the many variety of threading libraries available. Compilers supporting Erbium intermediate representation can then lower process instantiation to its runtime implementation rather then to the architecture or operating system target threading library. As the Erbium processes are persistent (long-lived threads), such abstraction overhead is negligible when compared with the overall application execution time.

Processes are defined similarly to regular functions and process instantiation is syntactically very similar to function calls. Unfortunately, at its runtime implementation, processes cannot be defined and instantiated as in the language definition. Code transformations are necessary through compilation to adapt initial definitions into its runtime support. Process code transformations are presented later in Chapter 4.

The process creation and instantiation at the runtime implementation is organized in four different steps: argument data structure definition, process instance creation, instance arguments initialization and process instantiation. Furthermore, any application instantiating Erbium's processes should wait for the termination of all instantiated and running processes, before its termination.

Erbium process runtime implementation does not differ from the most common threading systems available. As most thread creation runtime libraries, process functions should be defined having a single argument function, being executed based on a threading library defined function (*pthread_create* in POSIX) to which the process function is passed as argument. Erbium process creation is abstracted further by introducing a process instance data structure that contains the argument data, the process function pointer and any other process instance options relevant to the process creation. In any case, the limitations introduced by the Erbium runtime are eased by the *process instance* data structure, merging all the required information in a single entity.

The helper function *alloc_process_instance* simplifies the process instance creation by allocating the instance data structure, storing a pointer to the process definition and allocating memory for the process arguments, or any other necessary data related to the threading

system.

Apart from the most relevant Erbium related features, the instance data structure also provides support for threading system properties. For example, within POSIX threads, it is vital to keep a reference to a `pthread_t` data structure. Moreover, Erbium data structures can store optimizing attributes, such as process core pinning, scheduling policies, or any other target threading system configurable parameters. The libER implementation makes no effort to detect better scheduling or any other threading system properties for individual processes and uses the best known global options for the system. In order to achieve portability, the most common threading properties can be abstracted by the runtime. In target systems where such properties do not make sense, such implementations could be defined as empty.

Once a *process instance* is created, the memory region for the arguments is filled with relevant data. A common case is to cast the allocated memory into a specific data structure defined for the process arguments, facilitating arguments data access. The process code casts its argument (defined as *void* $*$) to the particular arguments data structure.

Once the *process instance* arguments and threading properties are set, one can execute the process by calling the runtime function *run_process_instance* which, in a POSIX implementation, uses *pthread_create* both with the callback and parameters memory reference. To keep track of all the executing processes, the executed processes instances are stored in a linked list, used by the *wait_process_instances_end* function during application termination, guaranteeing no process (thread) is alive at the main application exit.

In a POSIX implementation *wait_process_instances* is a traversal through all the *process instances*, executing *pthread_join* with each of the thread instances, waiting for each thread (process) to terminate. Once unblocked from each *pthread_join* call, the *process instance*, and its arguments memory reference are deallocated.

Figure 3.4 is an example of the necessary steps to perform a runtime process creation and to guarantee all processes termination.

```
1  void A(struct A_param *data) {
2    /* ... */
3  }
4
5  ProcessInstance instance =
6      alloc_process_instance(&A, sizeof(struct A_param));
7  struct A_param *param = instance->arguments;
8  param->rec = REC;
9  param->value = 1;
10 run(instance);
11  /* ... */
12 wait_for_instance_list_end();
13 return 0;
```

Figure 3.4: Process creation example runtime code.

LibER implements process instantiation using POSIX pthreads, making it available with all its supporting operating systems and target architectures. Depending on the actual threading library or even hardware support threading library, the complexity of the implementation might vary. Nevertheless, the Erbium runtime process creation abstraction should allow full support to any kernel-level threading system. User-level threads require much more complex transformations from the compiler perspective and, although very relevant and studied, user-level thread were not implemented.

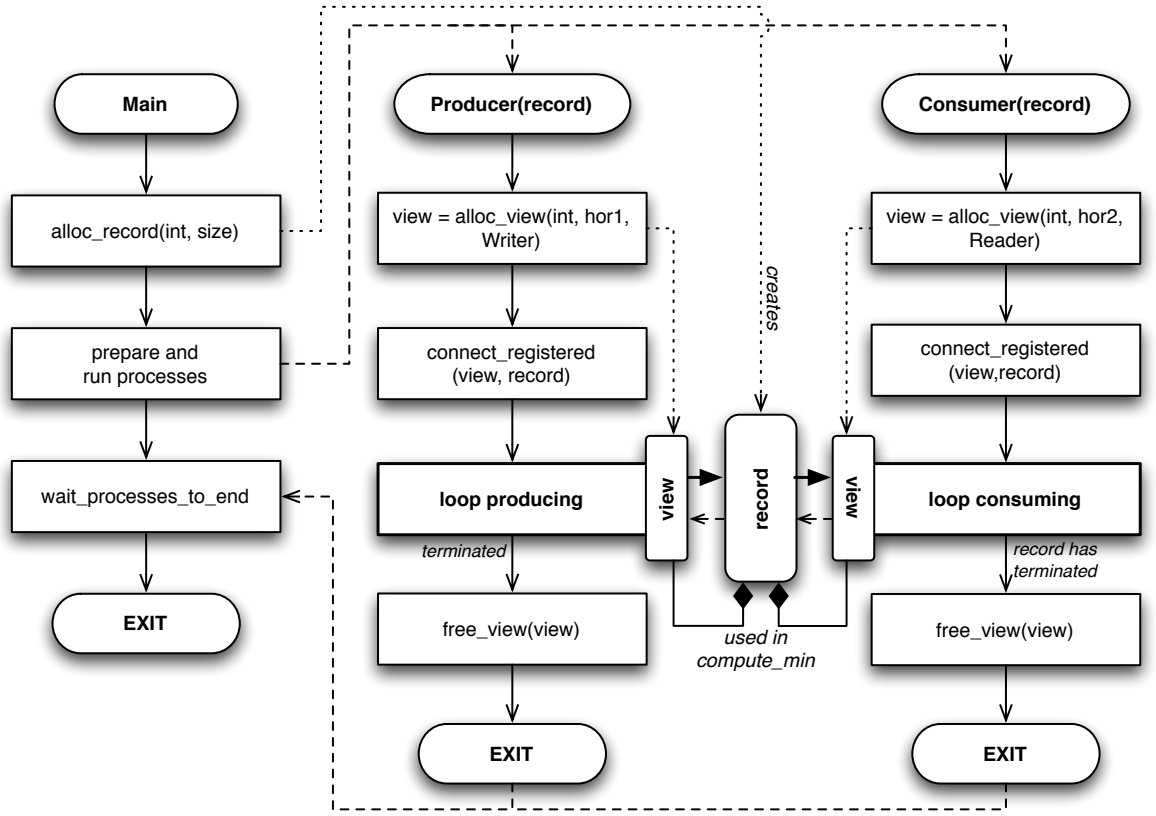


Figure 3.5: Simplified state diagram of a simple producer and consumer application initialization and termination interactions.

3.1.3 Record and Views

To allow a quick initialization and process execution start, records and views must be initialize as efficiently and independently as possible. Although synchronization calls are currently defined with no atomic instruction and fully lock-free, special care is necessary while designing initialization and termination. Without it, it is possible to very easily invalidate record data structures during initialization or termination, possibly breaking any of the properties supporting libEr atomic operations and memory barriers free implementation. This section details records and views initialization and termination in the perspective of Erbium's runtime support and its nuances.

Figure 3.5 is a outline diagram demonstrating a producer and consumer record and views initialization and termination. It is composed of a *main* (left column) function that defines a record instantiates two processes: a producer (middle column) and a consumer (right column). Function entry and exit points are represented with rounded shapes and the different relevant function stages are the squared boxes. Strong vertical lines represent the sequential function flow. Process initialization and termination is represented with the strong dashed lines. Small dashed lines represent data structures (record and view) creation actions.

In this example the application `main` (left column) starts by allocating a *record* data structure capable to store *size* elements of type *int*. The remaining of the `main` function refers to process execution as explained in the previous section.

Right after process execution, each process creates its own private view using *alloc_view*. It involves defining both a *type* for the view elements (first argument) and the view *horizon* (second argument). The producer process connects to the record, provided as parameter, using *connect_registered* runtime call. This call implies to assign the record to the view data structure and add a reference of this view within the record writers list. This list allows processes to perform the minimum index computation for all the writer and reader views, as mentioned in the previous section. Similarly to the producer process, the consumer also connects to the record, but this time, adding instead its view reference to the list of readers. In the diagram, the record list is marked by a line with a filled diamond (♦) meaning the record contains references to views. This list of views is previously mentioned in the context of the *minimum computation* and its implementation is further explained through the section.

At this point, both processes have connected views. View and record communication is represented through its connecting arrows. Strong arrows represent *commit* and *update* synchronization interaction as well as the data communication associated with this primitives. Dashed arrows refer to back-pressure, more precisely *release* and *stall* synchronization primitives.

Once the producer stops producing elements and wishes to terminate, it should call $free\_view$, specifying it no longer uses the view. In this example, the process does not perform any hand-over and terminates specifying no other process substitutes this connection. Since the record is created for a single registered writer view and having no other connected writer views, the reader view is notified of its termination once it reaches the index the writer view has last committed. The *update* primitive returns a index lower than requested, detecting the record termination (zombification) by calling *compute_min*.

Considering the record has no writer views connected, it is assumed as terminating and eventually the consumer process view realizes it exiting its working loop.

Similarly to the producer, the consumer calls `free_view`, verifying if anyone is still consuming from the record and performing the deallocation of all the record previously connected views, as well as the record data structure.

Record and view allocation

Chapter 2 defined both records and views as special language types, giving the misspercepction that both definitions are statically allocated. Counter intuitive to what was previously presented, at runtime support, both record and view data structures are defined and allocated dynamically through private constructor functions.

Record allocation is performed using *alloc_record* providing the size for each event data element (stored in its buffer) and the number of events initially allocated (size of the buffer). Buffer sizes should be either statically predefined by the programmer or predetermined during compilation based on compiler analysis, taking in consideration all of its connecting views horizon. The record allocation starts by allocating a buffer (with the size provided as argument) and initializing its upper and lower indexes to 0.

Furthermore, the lists of connected views (readers and writers) are also initialized in *alloc_record*. The linked-list implementation should mimic a traditional linked list allowing minimum computation to traverse all its view elements.

Views are created using *alloc_view* and are passed with a buffer type size, the horizon, required for either static or dynamic buffer size calculation, and a type, specifying the type (reader or writer) for the particular view. Although the *alloc_view* prototype is consistent

with the language definition, not all its parameters are used in libEr. The view data structure as presented does not store the buffer type size and the record buffer size is not dynamically verified against the horizon size provided. Still, in a distributed memory architecture, buffer type size and horizon arguments can be used to simplify the process local buffer allocation, allowing compilers to statically decide on the local buffer sizes without resorting to runtime communication with the record data structure. Both the *index* and *record_index* view attributes are initialized to 0 during *alloc_view*.

Algorithm 6 contains the definition for both *alloc_record* and *alloc_view* as defined in libEr.

Algorithm 6 Record and view allocation

```
1: function ALLOC_RECORD(buffer_type_size, buffer_size)
2:     r ← new Record
3:     r.buffer ← new char[buffer_size * buffer_type_size]
4:     r.upper_index ← 0
5:     r.lower_index ← buffer_size
6:     r.readers ← new ViewList
7:     r.writers ← new ViewList
8:     return r
9: end function

10: function ALLOC_VIEW(buffer_type_size, horizon, type)
11:     view ← new View
12:     view.index ← 0
13:     view.record_index ← 0
14:     view.type ← type
15:     return view
16: end function
```

View connectivity to the record

As explained in Chapter 2, Erbium supports two types of view connectivity, more precisely registered and non registered. Each is implemented resorting to two distinct functions, *connect_registered* for registered connections and *connect* for non-registered connection.

As previous mentioned, all the connected views are associated with the record through the insertion in the two distinctive readers and writers lists. Lower and upper record indexes are computed based on the traversal of both the lists computing the minimum index from all the connected views.

Connecting and disconnecting views in libEr is analogous to insert or remove view instances from those linked lists.

Considering connections occur concurrently, special care is required not to invalidate the linked lists. Concurrent manipulation of linked lists is a well known and studied problem as presented in [28, 54, 82] where lock free implementations of linked lists are presented, also explaining how and where atomic operations required.

In order to keep synchronization cost to a minimum the view connections cannot imply any extra complexity for the minimum computation algorithm. Although newly inserted elements in the list are eventually perceived by the minimum computation algorithm, list elements removal requires special care considering the linked list element can still be accessed

by a concurrent process. Luckily enough, linked list traversal is not affected by this operation as long as the view data structure is not immediately deallocated once removed. Moreover, the removed view element should keep its original forward linked list connectivity (*next* pointer), guaranteeing a correct minimum computation for the concurrent processes. In more detail when a view is removed from the list, it should keep pointing to the same *next* linked list element. Cleaning such forward pointer could possibly compute a higher incorrect minimum index value, considering its pointed *next* views would now be ignored.

In order to simplify complexity of the presented connectivity algorithms, the linked lists are manipulated resorting to the following concurrent safe linked-list functions:

- $insertView(list, view)$ inserts a view at the end of the linked list,
- $substituteView(list, view\_old, view\_new)$ substitutes the *view_old* in the *list* by the *view_new* content. *view_old* is not directly deallocated and the *next* pointer is left pointing to the same node.
- $deleteView(list, view)$ deletes the *view* argument from the list.

Apart from this mutation operation, the following search operation is also used:

- $findNext < TYPE > (list)$ is a polymorphic definition that per execution returns a non repeatable element of $< TYPE >$ type from *list*, or $NULL$ if there is no new fake views returned.

During initialization, a record should specify the amount of initial registered views connecting to the specific record. In libEr, similarly to the language definition, the function call *add_registered_views* allows to define it, taking both the number of expected registered reader and writer view connections. The libEr creates a fake view per number of requested registered views and inserts those views into the respective reader or writer list.

Algorithm 7 Add registered views to the record

```
1: function ADD_REGISTERED_VIEWS(record, nr_readers, nr_writers)
2:     for i ← 0..nr_readers do
3:         insertView(record.readers, new FakeView(record, Reader))
4:     end for
5:     for i ← 0..nr_writers do
6:         insertView(record.writers, new FakeView(record, Writer))
7:     end for
8: end function
```

Algorithm 7 shows the pseudo-code for *add_registered_views*. The $newFakeView$ function is the constructor for a new clear view that is never used by any process. Its purpose is only to enforce all the consumer processes to block in update primitive call while not all of the fake views are substituted by real views, i.e., all the expected views have connected. The fake view index is instantiated with the content of *upper_index* or *lower_index* record attributes depending if the view is of type Writer or Reader, respectively.

Once the record has specified the number of registered views, it is ready to accept connections. Registered connections involve requesting the record for a private *view connection identifier*. It is done using *get_new_view_id*, returning a single view connection identifier,

Algorithm 8 Get a non used view connection identifier

```
1: function GET_NEW_VIEW_ID(record, type)
2:     list ← getListforType(record, type)
3:     fake ← findNextFakeView(list)
4:     if fake = NULL then
5:         return NULL
6:     else
7:         return &fake
8:     end if
9: end function
```

later used by the process. In libEr, a view connection identifier is defined as a pointer to a linked list view node.

Algorithm 8 presents a simplified implementation for the *get_new_view_id* function, returning the address of the next $FakeView$ for the respectively typed *record* list. In order to guarantee the non repeated return of the same fake view, the access to the list through *findNextFakeView*, must execute atomically with respect to any other view mutation operations. Once no more fake views are available in the list, the function returns null.

The function $getListForType(record, type)$ is an helper function returning either the readers or writers list from the *record*, depending if the *type* argument is either *Reader* or $Writer$, respectively.

The function *connect_registered*, as explained in Chapter 2, connects a view to a specific record within a view connection identifier. In libEr, the view record pointer and its indexes are set by copying the previous view info (indexes), connected to the same view connection identifier associated view (initially a fake view) into the new connecting view. Once its attributes are set, the view is inserted in the respective record list, substituting one of the fake views. Algorithm 9 presents the prototype implementation of the *connect_registered* function.

Algorithm 9 Connect a new registered view or reconnect view to existing id

```
1: function CONNECT_REGISTERED(view, record, id ← NULL)
2:     list ← getListforType(record, view.type)
3:     if id = NULL then
4:         id ← get_new_view_id(record, type)
5:     end if
6:     if id = NULL then
7:         return NULL
8:     end if
9:     view_old ← (view) * id
10:    view.record ← record
11:    view.index ← view_old.index
12:    view.record_index ← view_old.record_index
13:    substituteView(list, view_old, view)
14:    return id
15: end function
```

Comparing with registered connections, non registered connections do not get initialized with the indexes of any previous connected view but rather the current indexes from the

record. The view is added at the end of the list and no view connection identifier is created. Algorithm 10 presents the algorithm for the "*connect*" function.

Algorithm 10 Connect a non registered view

```
1: function CONNECT(view, record)
2:     list ← getListforType(record, type)
3:     view.record ← record
4:     if type = Writer then
5:         view.index ← record.upper_index
6:     else
7:         view.index ← record.lower_index
8:     end if
9:     view.record_index ← view.index
10:    insertView(list, view)
11: end function
```

Once connected, views are ready for the processes manipulation, calling record synchronization primitives (manipulating the view sliding window limits), and eventually access data from the visible record events.

Termination and view hand-over

Record termination is the result of the disconnection of all the reader and writer views. All views are explicitly disconnected and terminated using the *free_view* function, called when the process no longer requires to access the view.

free_view expects a view argument and a boolean, specifying if the process is either permanently disconnecting the view or if it is performing a view hand-over.

Algorithm 11 Free view algorithm

```
1: function FREE_VIEW(view, terminate)
2:     if terminate is true then
3:         list ← ListForType(view.record, view.type)
4:         list.atomic_remove(view)          ▷ Remove view element from list.
5:     else
6:         Nop()    ▷ Do nothing. Some other process will substitute this view in the list
7:     end if
8: end function
```

When *free_view* is executed with *terminate* = *false*, the view is expected to be part of a process hand-over. In libEr implementation, the view is left untouched by *free_view*. If *terminate* = *true*, *free_view* will, atomically to other mutating list operations, remove the view data structure from its respective view list.

Apart from this termination scenario, processes might want to hand-over its view connection identifier to a different process, allowing some other view (process) to continue its work consuming or producing elements. In such cases, the process should free the view, but this time the view identifier is kept alive. In case of a process hand-over, the freed view is not removed from the connected views lists, enforcing process synchronization (minimum computation) to take its index into consideration. Once the hand-over process is instantiated, it should connect the new view using the same view identifier, substituting the freed view

in the synchronization. The connection is created using *connect_registered*, occurring in the same way as the previous explained fake view substitution.

Process hand-over implies view connection identifiers are unique and never reused. In this implementation, view connection identifiers are defined as pointers to views, which the programmer has no control over re-usability, i.e., memory allocation is maintained by the operating system. In any case, is is sufficient that the uniqueness holds between connection hand-overs.

Record zombification

Once all the writer views of a record get permanently disconnected (freed), the reader views should notify that no new indexes are available. When it happens, the record is considered a zombie, and future calls to *update*, when requesting an higher index value, return the maximum available record *upper_index*. Once *update* returns an insufficient index, the consumer processes must finalyze any remaining work using the still available events data (up to the record *upper_index* attribute) and possibly free the respective reader view.

In libEr, zombie records are detected by the absence of writer processes in its linked-list. Once the list gets empty, the record is marked as zombie and no future writer views are allowed to connect.

Algorithm 12 presents the zombification function for libEr. Notice that *is_record_zombie* was previously used in Algorithm 3.

Algorithm 12 Verify if record is zombie

```
1: function IS_RECORD_ZOMBIE(record)
2:     return record.writers == ∅
3: end function
```

Record and view deallocation

The process synchronization in libEr occurs concurrently to the record and view initialization and termination. On the other hand, connecting or disconnecting views to the same record (writers or readers list) requires a special care, thanks to the linked list data structure properties and restrictions.

Moreover, as the process synchronization occurs independently from the list manipulations, views cannot be immediately deallocated after the linked list removal. It can occur that a concurrent process is traversing the respective linked list having already reached the element for removal. If the view is as well deallocated, such pointer would become invalid and, when accessed, it could possibly access invalid data or even an out of bounds memory position, resulting in a segmentation fault. Moreover, if the minimum computation reaches an already removed view, the linked list traversal must not terminate in this view node, but rather proceed as it would if the view was not removed.

A simple and efficient solution to these problems is to postpone the view deallocation and to allow the freed view to follow up to previous *next* list element (in minimum computation list traversal). In other words, when the view is removed from the list of views, the attribute *next* is not cleared. Removed views are added to a deallocation list. During idle CPU times, views can be removed and deallocated from this list, if there *index* attribute is smaller than

the current record *lower_index* minus the buffer capacity, in case of reader views:

$$view.index < view.record.lower\_index - view.record.buffer.nr\_elems$$

For writer views the same condition also is valid, however using the *upper_index* instead of *lower_index*. During a record dealocation, if any disconnected views are still in memory, such views can be immediatelly deallocated.

The record deallocation executes once the last of its connected views is freed.

3.2 Wrap around indexes

Current generation of architectures support very big integer operations, even if composed by smaller sized operations. Such operation composition is performed by compilers generating extra instructions to address the complexity of the operation, in such cases this big integer variable type cannot be used as an Erbium index. Even if the architecture does support big single clock-cycle integer writes, the executing application could require to be active for days or even years, eventually overflowing the index variable. An example of such devices could be routers or television set top boxes.

As indexes in Erbium are assumed as monotonic, overflowing indexes should be taken into consideration when designing the runtime support for the language. Throughout language design, it was clear that such scenario would need a solution that did not degrade the synchronization overhead.

Index overflow is not taken into consideration in current implementation and invalidates libEr synchronization runtime implementation. For example, lets assume a single producer/-consumer example where the producer commits by an overflown index value. The consumer while still working with a non overflowed index expects a much bigger index from the minimum computation. Nevertheless, the minimum computation returns a too small value, not sufficient to validate the *update* condition, forcing the consumer process to deadlock, eventually deadlocking all the dependent processes and application.

In this scenario, it is clear that the current solution does not provide correct results and both the synchronization function primitives (*stall* and *update*) and minimum computation must be aware of wraparounds (overflows) and must return a revised minimum index of all the views.

The presented solution to this issue involves changing both the minimum computation (upper and lower indexes) as well as the update and stall functions. The new algorithms are presented in Algorithms 13 and 14.

Back in Algorithm 3, the update blocks the process execution using a while loop, verifying if the view *index* attribute is greater or equal to the current cached *record_index* (the condition is negated in the algorithm because it is used in a *while* loop). This condition, is one of the problems associated with overflown indexes.

One possible approach, valid for overflows, is to check if both the index variables have the same sign[1] (the same most significant bit), in which case the comparison in *update* and *stall* is always valid. By dividing index ranges in two distinct (positive and negative) regions, we also introduce undefined conditional regions. From previous understanding, we cannot compare those values using less than or greater than operators. As those values are

[1]This section refers to sign as the most significant bit of any integer type. Please notice that variable type signing is irrelevant runtime execution of the presented algorithms.

in different regions it is certain both will contain different values, i.e., one is bigger than the other.

In an overflow, such detection can be performed based on the assumption that both values should be near a sign transition (either 0 or at an overflow value). The monotonically smaller index when added by 1/4 of the index type range should overflow or cross by 0. Using this technique it is possible to detect which of the indexes has a lower value, without relying on any of the standard, non working, greater than or less than operators.

More precisely, when comparing different sign indexes:

$$view.record\_index > index \iff sign(index + (1/4 \times range)) = sign(view.record\_index)$$

Commit and release should also take into consideration overflown indexes, more precisely since its current implementation compares indexes with forbidden operators (less and greater than). The comparison is used to verify if the latest index is bigger than previous committed index, enforcing monotonicity of the view index. Such condition is not mandatory as the process code should perform commit and release with monotonic indexes, guaranteeing the view monotonicity.

Algorithm 13 Update and release with indexes wraparound

```
1:  function UPDATE(view, index)
2:      iteration ← 1
3:      while (asSameSign(view.record_index, index) && view.record_index < index) ||
4:          (!asSameSign(view.record_index, index) &&
5:          sign(view.record_index) = sign(index + (INDEX_RANGE/4))) do
6:          if iteration > 1 then
7:              if view.record.writer = ∅ then
8:                  return view.record.upper_index
9:              end if
10:             sleep()
11:         end if
12:         view.record_index ← compute_min(view.record, Writers)
13:         iteration ← iteration + 1
14:     end while
15:     return index
16: end function
```

Although update and stall are now validated for wrap-around indexes, the minimum computation is also comparing indexes without validating its comparison. In order for it to take overflows in consideration, the minimum computation should also use the sign of the indexes when computing the minimum. To fix for wrap around indexes, the minimum computation implementation should have two different computation states. One state computes the minimum of the positive indexes and the other computes the minimum of the negative indexes. The state transaction occurs when the current state contains no valid indexes, or in other words, no index is within the state index ranges (positive or negative value). Right after a state transaction, the list of views should be traversed once more to compute the minimum for such indexes state. Algorithm 14 presents the wraparound minimum computation implementation.

Algorithm 14 Minimum computation with indexes wraparound

```
1:  function COMPUTE_MINIMUM(record, type)
2:      counter ← 0
3:      min ← ∞
4:      list ← ListForType(record, type)
5:      state ← StateForType(record, type)
6:      for view ∈ list do
7:          if (*state = 0 && view.index ≥ 0) || (*state = 1 && view.index < 0) then
8:              min ← MIN(min, view.index)
9:              counter + +
10:         end if
11:     end for
12:     if counter = 0 then
13:         swap(state)
14:         goto line 6
15:     end if
16:     return min
17: end function
```

Wraparound safe processes

Code generators or/and programmers must take index overflows into consideration in order to generate wraparound safe processes. Please consider the code example in Figure 3.6. The example on the left suffers from the same problems as the update and stall implementations presented earlier, more precisely on the comparison with the return from update to identify record termination.

```
1 while(1)
2 {
3   ret = UPDATE(v, i +10)
4   if(ret < i +10)
5     break; // terminate
6   ...
7 }
```

```
1 while(1)
2 {
3   ret = UPDATE(v, i +10)
4   if(ret != i +10)
5     break; // terminate
6   ...
7 }
```

Figure 3.6: Code examples between a non safe (left) and safe (right) wrap-around process.

By comparing two indexes with less than (`<`) operator, one is assuming that the application will never overflow its indexes. Similarly to update and stall primitives such comparison with overflown values might result in hard to detect deterministic deadlocks. To avoid such caveats as a first rule, processes should only compare indexes using equality or inequality operators (`==`, `!=`). Considering a compiler implementation of Erbium language, the compiler should guarantee the correct convertion of any index comparison, possibly using specific runtime calls, validating at an wrap-around safe level the result of such comparisons.

Apart from problems within indexes comparisons, it might occur that some buffer implementations also do not support index overflows. Please consider the presented circular buffer libEr implementation where the index buffer position is computed based of the *modulo* of its size. When an overflow occurs, the result of the modulo computation is not guaranteed to provide a continuous position in the buffer. Let's assume an index type range of $[0, 2^{32}[$, which means that when such monotonic index reaches 2^{32} in reality it

becomes 0. In order for the buffer index computation to be correct and provide continuous buffer positions, both the buffer index computation for 0 and 2^{32} should be the same ($0 \ \% \ buffer\_size = 2^{32} \ \% \ buffer\_size$). As an example, if the buffer size is 10, the index computation for 0 and 2^{32} give different results ($0 \ \% \ 10 = 0$ and $2^{32} \ \% \ 10 = 6$). A simple solution to the existing buffer implementation is to always create buffers of size powers of 2. Index overflows always occur at powers of 2, forcing buffer index computation to always return consequent buffer positions when overflown.

3.3 Supporting other runtime environments

Presented libEr implementation exploits x86 properties, reducing synchronization overhead up to its minimum. Unfortunately, it is restricted to x86 architectures and is power inefficient and non-optimal, in unbalanced or over partitioned applications. By creating libEr without any atomic operations, we also restricted such version from being lazy in respect to its synchronization primitives implementation.

Nevertheless, there are many non explored implementation options, each with its benefits. For example, increasing application performance, supporting a different target architecture or reducing execution cost.

Apart from obvious reasons such as architecture memory model, which has a direct impact on the Erbium runtime implementation, there are also many not so noticeable target architecture properties that may enforce different implementations. An example of such properties are, for example, differences in the memory consistency model, cache coherence or even in the instruction set architecture implying a different strategy when implementing the runtime support. Moreover, it can also occur that the target operating system does not support previously used libraries.

The presented implementation of the libEr uses the POSIX threading library for the thread creation and exploited the current generation X86 architecture properties avoiding the usage of lower level synchronization primitives. Apart from the memory fences and atomic instructions free implementation, libEr was also implemented with more lazy synchronization using POSIX library mutual exclusion primitives and lower level *Futex* [27] primitives. Nevertheless, neither *Futexes* or POSIX runtime library are available to all operating systems.

Not only compatibility should be taken in consideration. Any runtime implementation must be able to efficiently execute its target applications. Depending on the application parallelism level, data communication and its granularity, different applications can have very distinctive execution times. Moreover, performance must not be the only decision factor. A single application code can be optimal in all respects with a specific runtime implementation, but be a total disaster when using a different implementation. Applications, runtime supports and target architectures are entangled and can never be considered optimal unless analyzed as a whole.

The current section presents several of the possible implementation details taking in consideration all the previous mentioned criteria.

3.3.1 Threading systems

The libEr process creation is implemented using the Linux kernel-level threading system, more precisely using the POSIX library.

Although not implemented, the Erbium language is compatible with user-level threading. Extra compilation code transformations are needed, adapting process code to the target user-level threading library requirements. Examples of such adaptations are to explicitly backup process state in case of any user-level context switch.

User-level threads provide a runtime control over scheduling, allowing the runtime support to explicitly decide when the processes "yield" each thread, i.e., decide when to perform a context switch. As a streaming language, Erbium's inter-process dependencies are very clearly defined. Once locked, their threads do not get unblocked just by retrying but only if some other process thread commits or releases events. The kernel-level thread scheduler assumes a fair distribution of clock cycles between all the threads and does not take into consideration any possible thread dependencies, as is the case of Erbium's processes.

To minimize the execution time, CPU cores must never be idle or spend cycles, as long as there is no other process with a higher probability to work.

User-level threads allow better control process execution switch and keep executing the same process while it has work. Such approach reduces the number of context switches, considering no other process will "steal" clock cycles. Moreover, this implementation is also good in respect to cache effects, minimizing cache misses. By better controlling thread context-switches, allowing processes to execute for longer periods, it improves data-locality and allows processes to hide memory transfers through cache pre-fetching.

Mostly beneficial cases for user-level threads are the applications which cannot be partitioned and balanced for the exact number of hardware-threads available in the system. As user-level threads require a user-level scheduling approach, such scheduler, instead of blindly executing one of the process (or the oldest sleeping), could also verify for an executing condition (update and stall conditions) and perhaps even execute the minimum computation for the implied list of views. By doing so, it would reduce the chance for bad context switches, where the process cannot execute and immediately initiates another context switch.

3.3.2 Synchronization

The presented libEr implementation is the simplification result of a previous multi-architecture shared memory lazy implementation. In this section, a possible reverse path is taken, briefly explaining the differences to the current presented libEr implementation, together with the difficulties associated with such a lazy implementation.

Busy waiting and deadlock freeness

In libEr, concurrent calls of the update or stall primitives create concurrent conflicting calls to minimum computation, generating execution races as they write the record shared indexes. The main concern with this approach is the fact that "losing" the race is in fact "winning", i.e., the slowest process is the one writing its computation for the last time. Most likely the minimum computed index is also smaller than the previous written index, and although it does not invalidate the index value, it breaks monotonicity of the index variable.

Breaking monotonicity of the record upper and lower indexes is not a problem when the runtime system is able to detect and recalculate it. Without the detection and recalculation, the system is vulnerable to deadlocks, considering the index gets smaller (not monotonic), possibly not high enough to satisfy the respective primitive call to unblock.

The libEr implementation solves this problem by implementing a busy waiting strategy

which, together with the POSIX kernel-level threading, enforces an eventual context switch to any previously blocked thread (process) where a new minimum computation is executed. Using user-level threads, such solution can also be made valid. One might consider that such concurrent minimum computation execution is impossible in user-level threads. However, please remember user-level threads are executed "inside" concurrent kernel-level threads (user-level threading space). Whenever distinct processes connecting to the same record data structure are scheduled in different user-level threading, concurrent execution of minimum computation is possible.

Although correct, busy waiting solution is not power efficient, considering:

- it loops continuously through all of the processes trying to identify which of the processes requires to execute,
- each individual kernel-level thread performs its own minimum computation, possibly invalidating record indexes monotonicity.

Implementing a more lazy synchronization approach implies to mitigate concurrent minimum computation executions.

Minimize concurrent writes to shared indexes

Algorithm 15 presents a solution by allowing only the first of concurrent computations to write over the respective index. Such implementation is based on a atomic *compare_and_swap* operation. Before the function starts traversing all the views searching for a minimum index, it stores the value currently set in the record index. This value is later used in *compare_and_swap*, which only substitutes the content of the index (1st. arg.) by the new index (3rd. arg.) if its current value is identical to the previous copied index (2nd. arg.).

Algorithm 15 Minimum record view index computation with compare and swap

```
1:  function COMPUTE_MIN(record, type)
2:      if type = Writers then
3:          list ← record.writers
4:          index_ptr ← &record.upper_index
5:      else
6:          list ← record.readers
7:          index_ptr ← &record.lower_index
8:      end if
9:      tmp ← ∞
10:     previous_index ← *index_ptr
11:     for view ∈ list do
12:         tmp ← MIN(tmp, view.index)
13:     end for
14:     compare_and_swap(index_ptr, previous_index, tmp)
15:     Returntmp
16: end function
```

This algorithm can result in the following scenarios:

- If some other process is performing a re-computation, and the index is eventually changed by the first succeeding one, the second one will not write in the respective record index, making its re-computation appear as invalid to the system.

- If for example the re-computation of the first execution brings no change to the record index value, but a further one does, then the value computed by the new call is accepted and updated.

This approach guarantees concurrent minimum computations are not taken into consideration, enforcing at all times the monotonicity of the involved index values. Without it, too many cache inconsistencies and invalidations occur forcing cache coherence algorithm to be triggered very often, slowing down the system performance [26].

Mutual exclusive minimum computation

Allowing an atomic instruction free implementation and concurrent minimum computation is useless and resource expensive. Whenever a process is computing an eventual invalid or not used computation, all the computation resources are wasted while it could be used executing relevant process code. In any case, guaranteeing mutual exclusivity of the minimum computation might in some cases be even worse. Such cost greatly depends on the running application, target architecture and operating system. Examples are the synchronization architecture instructions available, memory consistency and cache coherency models or the number of concurrent processes connected to a single record (multiple producers or consumers). Also, the more concurrent records are, the greater is the chance for concurrent minimum computations to occur and slowdown the execution.

In cases where it makes sense, an obvious solution is to only allow a single process to recompute the minimum simultaneously. Such behaviour can be implemented using *pthread_mutex_trylock* or for a possible faster solution using a shared boolean variable and an atomic compare and swap.

Processes not computing the minimum would sleep for a while, giving other waiting processes a chance to execute.

Algorithm 16 prototypes a mutual exclusive minimum computation call implemented in the update primitive.

Lazy waiting

Even after minimum computation mutual exclusivity, processes are still frequently awakened, verifying if *update* and *stall* should unblock. This verification might not seem expensive but, when considering the cost of thread context switching together with the possible high number of concurrent processes and insufficient number of cores, such overhead is "overwhelming".

In this case, instead of sleeping (context switch), each process can *wait* for a *wake* signal, announcing a possible unblocking opportunity. Waiting threads are flagged by the operating system thread scheduler, and so are scheduled for execution until the respective signalling data structure has received a wake signal.

Algorithm 16 is a non detailed prototype implementation of both *commit* and the non wrap-around *update* version using lazy waiting. In this algorithm, more precisely on *update*, instead of the sleep call producing an eventual thread context switch, the thread executes a wait on its record defined signal attribute. The signal primitives used in the algorithm are defined based on the Futex system calls synchronization primitives. Nevertheless, such primitives could be easily adapted to any signal capable synchronization library. *Signal_update* is initialized during record allocation, accordingly to the actual signal implementation. *Stall*

and *release* suffers similar implementation changes as presented in the *update* and *commit* algorithms.

Both *single_wake* as *broadcast_wake* are waking primitives, taking a reference from the signal description variable and triggering either the wake of a single or all of the waking processes, respectively.

The *wait* primitive takes two parameters, more precisely the same signal descriptor and the signal message id. The signal message id allows the wait primitive to identify if any wake has been executed after the provided message id was collected. In the algorithm, the message id is collected at Line 10. The *wait* primitive, at Line 8, enforces that any wake message occurring after this line is taken into consideration in the waiting primitive. Furthermore, if any other process performs a wake on this same signal, such wake will be taken into consideration and the wait primitive call will not block.

Let us suppose that while executing *update* and when trying to execute *compute_min*, some other process is already executing it, forcing trylock to reject its attempt to execute. Meanwhile, the other thread *compute_min* terminates and updates *upper_index*. While the process reaches the condition to wait, it should be able to verify if any other process has terminated a minimum computation. Right after terminating the minimum computation, the process executes a signal wake. This signal wake, as executed in between the condition check and the actual wait primitive, must ensure that the next upcoming wait is ignored. Once this wait is ignored, the *update* call re-iterates and verifies the newly minimum computation against its argument index.

The *commit* implementation is very similar to the already existing one but this time a single wake signal is generated after the view private index is changed. Such call wakes only a single process waiting for the same signal. This process most likely succeeds at *trylock* check and computes the respective record index.

Once the minimum computation concludes, it generates a broadcast message waking all the remaining processes.

Index based lazy waiting

Although lazy, the previous implementation does not independently wakes individual processes but rather wakes all the record connected ones. Independently of the computed minimum value, every consumer is wakened, forcing possible context switches that would verify for the *update* or *stall* unblocking condition and go back to wait state, i.e., computed minimum is not high enough for the required operation index argument. A precise implementation would only wake processes waiting for an index below the returned minimum. However, verifying for such cases would be very inefficient.

A possible half-way approach is to create a pool of signals and associate each signal with ranges of indexes. The blocked processes will select the right signal to *wait* depending on their *update* or *stall* index argument. *Commit* and *release* trigger a *wake* to a single process of each of the signals, of the respective index range. The woken process, after executing the minimum computation, broadcasts a wake messages to all processes waiting for this same signal, as well as all smaller index ranges signals.

This lazy implementation is similar to the previous presented one, although there is more than one signal per record index and for that matter the respective signal element in the array should be computed based on the index position.

This solution uses the same signal implementation as before, although requiring an extra

Algorithm 16 *Update* and *commit* primitives using lazy waiting

```
 1: function UPDATE(view, index)
 2:     iteration ← 1
 3:     while view.record_index < index do
 4:         if iteration > 1 then
 5:             if is_record_zombie(view.record) then
 6:                 return view.record_index
 7:             end if
 8:             wait(view.record.signal_update, signal_id)
 9:         end if
10:         signal_id ← *view.record.update_signal
11:         if view.record.upper_index ≥ index then
12:             view.record_index ← view.record.upper_index
13:         else                                          mutual exclusive
14:             if (trylock(view.record.mutex_update) == 0) then
15:                 view.record_index ← compute_min(view.record, Writers)
16:                 broadcast_wake(view.record.signal_update)
17:             end if
18:         end if
19:         iteration ← iteration + 1
20:     end while
21:     return index
22: end function

23: function COMMIT(view,index)
24:     if view.index < index then
25:         view.index ← index
26:         single_wake(view.record.signal_update)
27:     end if
28: end function
```

Algorithm 17 Mutual exclusive minimum computation update section with index lazy waiting

```
1: if (trylock(view.record.mutex_update) = 0) then
2:     init_signal_index ← signal_for(view.record.signal_update)
3:     view.record_index ← compute_min(view.record, Writers)
4:     final_signal_index ← signal_for(view.record.signal_update)
5:     for i ∈ {init_signal_index, ··· , final_signal_index} do
6:         broadcast_wake(record.update_signals[i])
7:     end for
8: end if
```

computation for the right index to wait or wake up, depending on the target index value. Moreover, the broadcast messages are now executed for all the signals associated with the index between the previous computed minimum index and the newly computed one.

Algorithm 17 presents the mutual exclusive minimum computation call differences from the previous lazy version, implementing index based lazy synchronization.

Although signal waiting avoids context switching to not yet available processes, signals also introduce significant overhead. As example, in *futex*es both wait and wake primitives are implemented as system calls. System calls have an initial call overhead associated with the operating system switch into kernel-mode.

In current implementation, each commit and release call performs a wake, regardless of the existence of waiting processes. This problem is dependent on the actual implementation of the signals. The *futex* implementation implies a per call overhead considering its system call based implementation. The *pthreads* conditional waits, on the other hand, provide the means to identify when some other thread is already waiting, resulting in minimal wake overheads considering its user-level definition. Nevertheless, *pthreads* conditional wait implies the usage of *mutexes*, resulting in an extra overhead.

3.3.3 Data communication

Shared memory architectures, although having a single common memory space abstracting the programmer from the data communication, have an independent memory space (cache) for each of its parallel cores. Each private cache is updated, respecting the architecture coherence and consistency protocols. Data communication through cache coherency is a consequence of the simultaneous data accesses by more then one of the processor cores, possibly resulting in data inconsistencies between the different private memory (cache) copies. Like any other memory operation, such transfer operations (resulting from coherence protocol) take many clock cycles when comparing with L1 cache and register copies. Although such operations execute asynchronously, processors can only use such advantage when future memory uses are anticipated.

Like in distributed memory architectures, shared memory architectures can also benefit from explicit communication primitives. In X86, the *PREFETCH* instruction initiates an asynchronous transfer for a particular memory region, if that region is not available or invalidated at the core private cache.

Erbium allows to specify which memory regions will be required for future process iterations, more precisely with the use of the *transfer* primitive, as explained in Chapter 2. Although not previously mentioned, the libEr implementation includes this primitive, anticipating future memory usage and initiating asynchronous copies to the local CPU cache. Prefetch execution occurs immediately after the minimum computation, even if the process is still executing previous indexed computations. The prefetch must take into consideration the layout of the architecture caches, allowing to better define the best amount of data elements to prefetch.

Depending on the actual application, it is possible that not all of the record events are used by all the record consumers, making the prefetch of all those non needed memory regions resources wasteful. The *transfer* primitive, by allowing to specify the precise required events allows to minimize such cost. Using the *transfer* primitive, only the indexes in its argument range are prefetched, anticipating the data necessary for the next process iteration, reducing cache misses and hiding memory latency.

Process pinning

Each time a thread is scheduled for execution, it might not be scheduled to execute in the same processor core and may thus be away from its previously used cache. When such migration occurs, it implies that all the necessary memory is transferred back to the new executing core low-level cache. Most applications, if well tuned or optimized, should at least make full use of the first level cache per process scheduling.

To minimize such costs, Erbium processors can be pinned to a specific cores. Nevertheless, pinning implies an initial knowledge of the available number of target hardware threads, apart from a good knowledge on how the application should be scheduled in order to maximize the processor usage. A bad initial pinning results in reduced CPU usage, traducing in slowdowns when comparing to dynamic partitioning [34].

3.3.4 Targeting applications and systems

No single runtime implementation is optimal for every application, operating system or architecture. Depending on the target system goals, the type of the application and the architecture properties, the runtime requires precise tuning. Moreover, there are many important details that must be taken in consideration when optimizing both application and runtime library support.

The libEr runtime support offers good performance when there is a good understanding of the application and the complexity of each of its processes, requiring well partitioned and balanced applications. When the number of processes matches the number of hardware threads, the processes almost never require to context switch. For a load-balanced application, synchronization most likely never needs to block.

If an application is not load balanced, too many cycles are spent spinning and verifying stall and update unblocking conditions, making such implementation not very power efficient.

In an application with more parallel processes than hardware threads, all the context switching and cache effects spend too many clock cycles. On the other hand, without as many processes, the full processing power is not used leading both to a non-optimal execution and power inefficiency.

Without a doubt, well partitioned, balanced and statically scheduled applications are the best option to achieve both a fast and power efficient execution [34]. Nevertheless, it is not always possible to produce such applications, or doing so would take too much man power. In such cases, reducing synchronization cost by introducing a lazy synchronization is the best option, forcing processes to wait, not allowing kernel schedulers to optimistically switch threads to such processes. Such an approach introduces extra overhead due to its required atomic and mutual exclusion operations. On the other hand, when comparing with the many worthless context switches of a busy-waiting implementation, such overhead is minimal.

User-level threading is another option, where context switches only occur when a process can no longer execute. Moreover, process scheduling is the result of the implicit context switch of a blocked process. Moreover, user-level scheduling can verify for executability of the process by, before hand, verifying for the update or stall primitive checks. Whenever a process starts to execute, it will immediate skip any verification and starts executing the next iterations. User-level thread solutions have the overhead of an extra scheduler algorithm, deciding which process should execute next.

3.4 Experiments

Experiments are based on the busy-waiting presented version of libEr without wrap-around support, targeting a 4-socket Intel hexa-core Xeon E7450 (Dunnington), with 24 cores at 2.4 *GHz*, a 4-socket AMD quad-core Opteron 8380 (Shanghai) with 16 cores at 2.5 *GHz*, both with 64 *GB* of memory, and an Intel quad-core Core 2 Q9550 at 2.83GHz. These targets are respectively called Xeon, Opteron and Core 2 in the following.

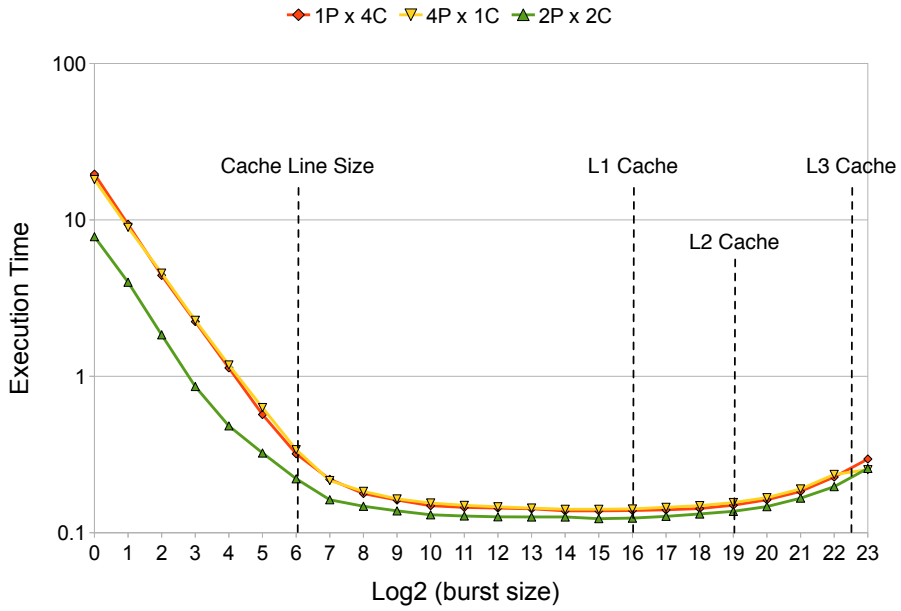


Figure 3.7: Burst size impact on Opteron.

One of the experimental studies is a synthetic benchmark called `exploration`, with multiple producers broadcasting data to multiple consumers. Each process implements a simple loop enclosing a pair of synchronization primitives updating, stalling, committing and releasing a burst of k indexes at every iteration. The workload for each index amounts to a single integer load (consumer side) or store (producer side) only.

Erbium ports of one classical signal-processing kernel and three full applications: `fft` from the StreamIt benchmarks [80], `fmradio` from the GNU radio package and also available in the StreamIt benchmarks, a `802.11a` code from Nokia [58], and `jpeg`, a JPEG decoder rewritten in Erbium from a YAPI implementation of Philips Research [78]. They are representative of data crunching tasks running on both general-purpose and embedded platforms. These applications are complex enough to illustrate the expressiveness of Erbium, yet simpler than complete frameworks like H.264 video that would require adaptive scheduling schemes not yet implemented in Erbium [14]. In addition, `802.11a` involves input-dependent mode changes that do not fit the expressiveness constraints of StreamIt, `jpeg` has variable computation load per macro-block and both `jpeg` and `fft` feature very fine grain tasks.

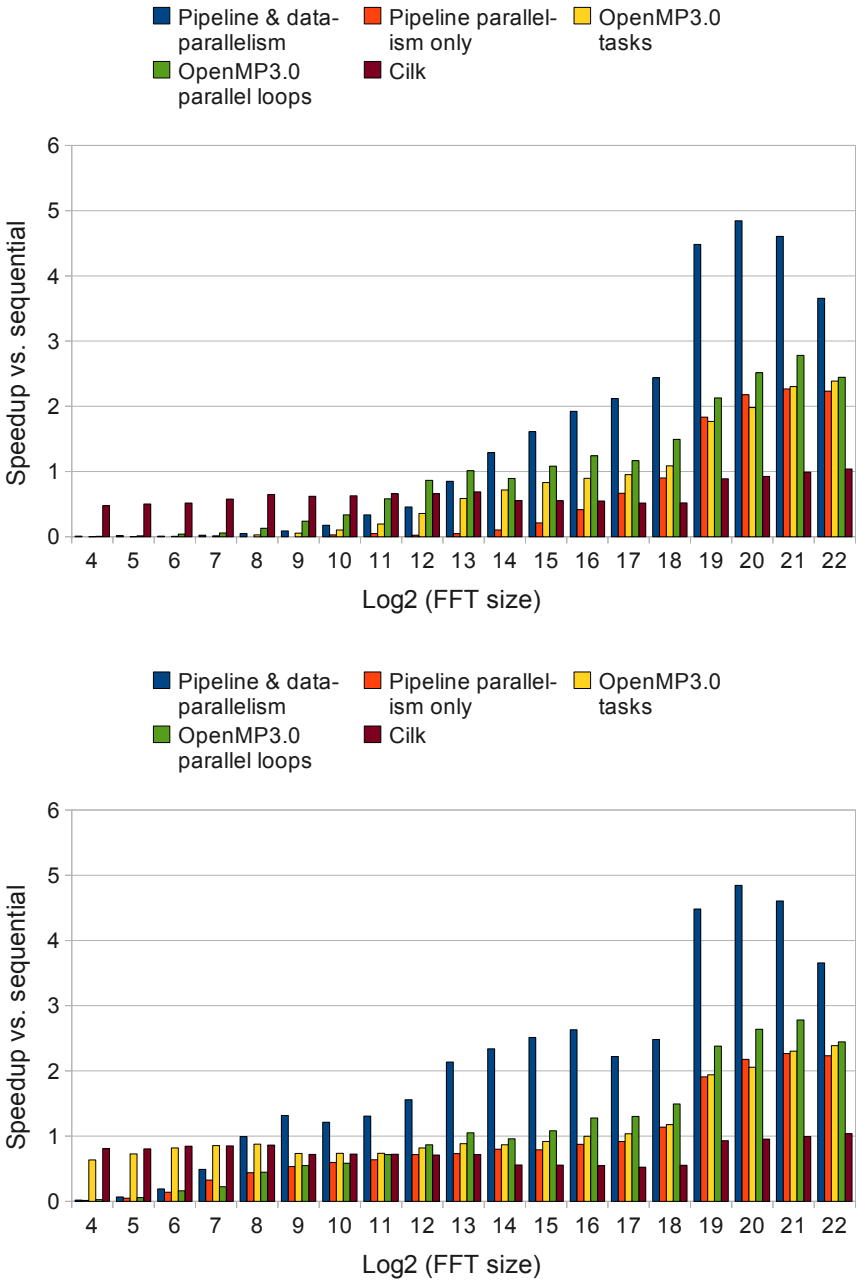


Figure 3.8: Performance of `fft` on Xeon (24 cores). Single settings (top) and best settings per data point (bottom).

3.4.1 Synthetic Benchmark

The synchronization overhead for the `exploration` benchmark is shown in Figure 3.7. Four configurations per target were tested: 1 producer and 1 consumer, and 1 producer and 4 consumers, 4 producers and 1 consumer, 2 producers and 2 consumers. All views of producer(s)/consumer(s) are connected to the same record structure with an horizon of 2^{24}

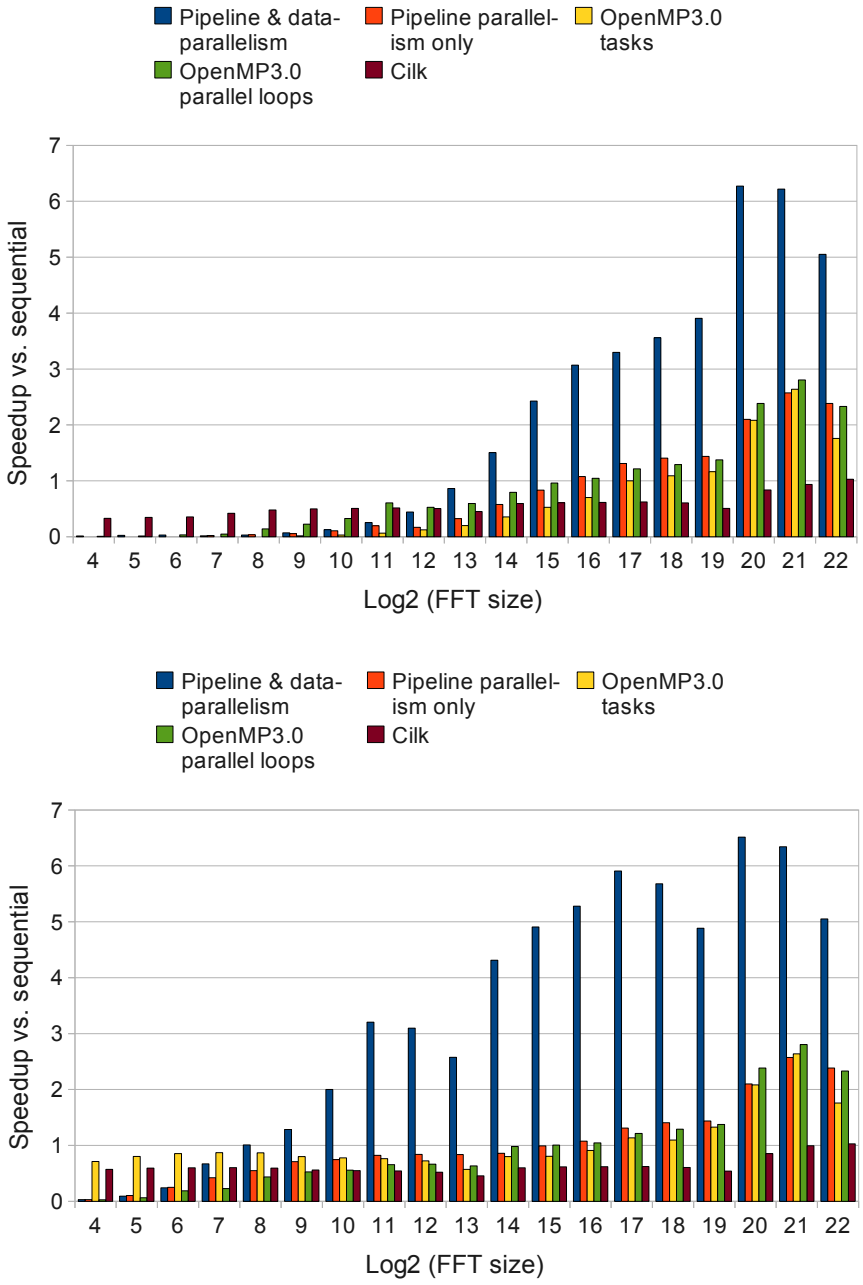


Figure 3.9: Performance of `fft` on Opteron (16 cores). Single settings (top) and best settings per data point (bottom).

elements. Each execution processes and communicates 2^{26} indexes. Threads are pinned such that producer(s) are all mapped to different cores of the same chip, and all consumer are mapped to different cores on a *different* chip.

False sharing induces severe overheads for tiny bursts and is the cause of the wide performance instabilities. The burst size remains an important factor passed the cache line size, but synchronization overhead becomes negligible for bursts of 1024 indexes or more. In addition,

the scatter and gather performance is also excellent: small bursts remain profitable even for complex configurations with multiple producers and consumers. This validates the choice of an expressive concurrent data structure: performance is excellent on a simple pipeline, while offering maximal flexibility to support combinations of pipeline- and data-parallel computations in a high-level language.

The single-producer single-consumer configuration reaches a maximum of $103M$ index computations per second on Xeon: on average 23.3 cycles per index, an order of magnitude shorter than the cache line transfer across x86 chips. This paradox is easily explained by the large-enough horizon, *update* and *stall* almost never result in a blocking synchronization, and by our cache-conscious algorithm amortizing cache line transfers over a large number of calls to the synchronization primitives.

3.4.2 Real Applications

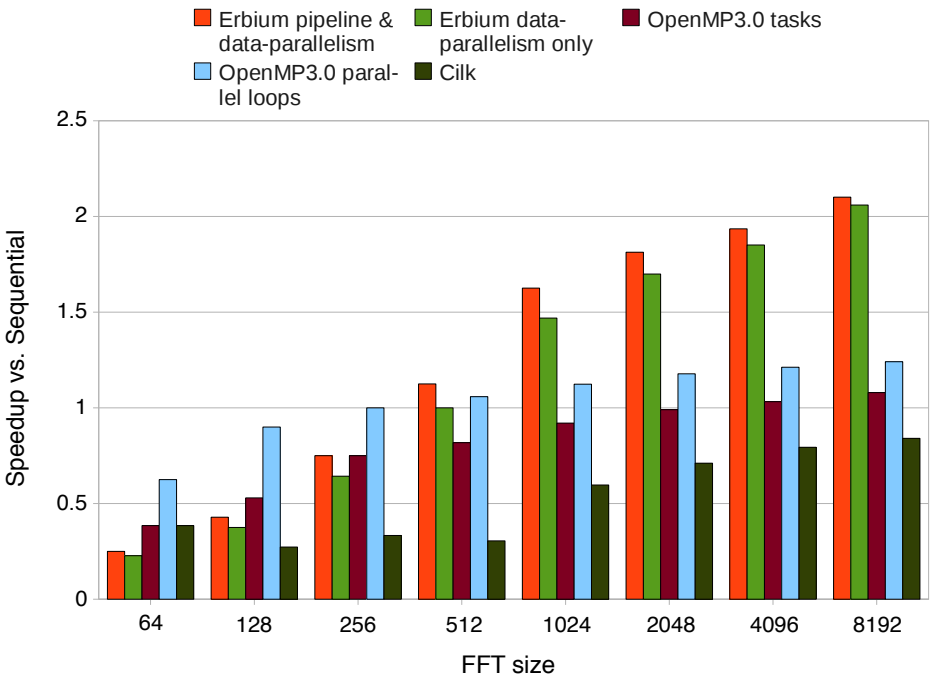


Figure 3.10: Performance of `fft` on Core 2.

Figures 3.8, 3.9 and 3.10 compare the performances of various parallel versions of the FFT kernel while considering multiple vector sizes. The baseline is an optimized sequential FFT implementation used as a baseline for the StreamIt benchmark suite. The first two bars are two parallel versions using Erbium, the next two bars are OpenMP versions, the last bar is a Cilk version. Combined pipeline and data-parallelism achieve the best speedups, compared to pure data-parallelism (both with Erbium). The size of the machines and the associated cost of inter-processor communication sets the break-even point around vectors of 256 single-precision floating point values.

FFT does not naturally expose much task parallelism because of the dependence patterns in the butterfly stages, yet pipelining computations across different stages remain possible, and favors local cache-to-cache communications over external memory accesses, explaining

the performance improvement of the combined version. On Xeon and Opteron, exploiting this pipeline parallelism allows to reduce contention, even if at the expense of some data-parallelism. To better analyze the intrinsic synchronization performance of Erbium, Figure 3.10 shows the performance results on the smaller single-node Core 2 platform. As the possible concurrency is reduced, the data-parallel versions stand more of a chance, with a shared L3 cache and L2 shared among each two cores. However, even in this unfavorable setting, the addition of pipelining gives enough edge to our combined data- and pipeline-parallelism approach, which outperforms the task-parallel and data-parallel implementations in OpenMP, as well as a Cilk implementation [30].

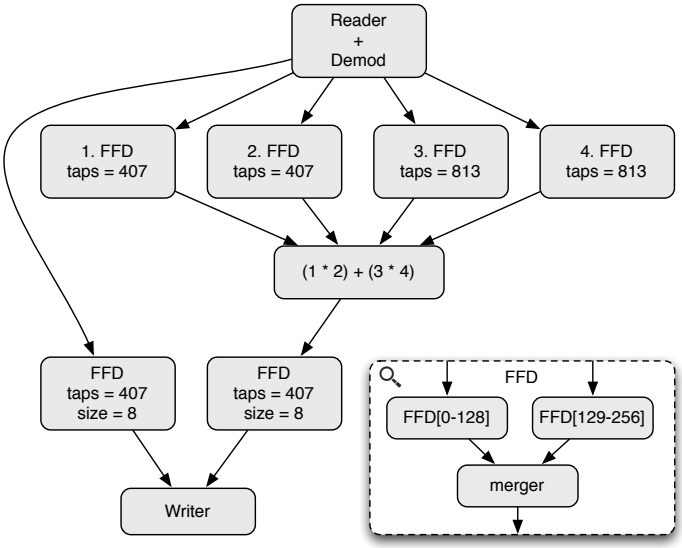


Figure 3.11: Informal data flow of `fmradio`.

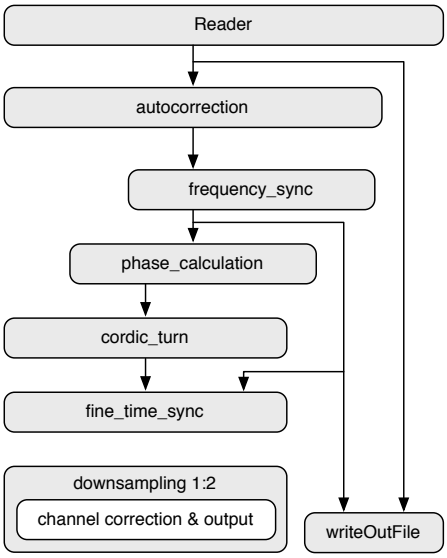


Figure 3.12: Informal data flow of `802.11a`.

Platform	Seq. `-O3`	`-O2`	`-O3`	`-O3` vs. `-O2`
Xeon (24)	1.14	10.1	12.6	1.25
Opteron (16)	1.52	9.51	14.6	1.54

Figure 3.13: Speedups results for `fmradio`.

Platform	Task-Level	Data-Parallel	Combined
Xeon (24)	1.85	1.84	6.67
Opteron (16)	2.73	2.81	7.45

Figure 3.14: Speedups results for `802.11a`.

Now considering the three full applications, `fmradio` and `802.11a` did not require any radical design changes to achieve scalable performance, while `jpeg`, written initially as a fine-grain KPN, required systematic conversion of the synchronization and communication methods.

On `fmradio`, exploiting task and pipeline parallelism is easy but shows limited scalability (6 concurrent processes). Exploiting data parallelism is not trivial and involves an interesting transformation. The original code uses a circular window using modulo arithmetic and holding the results of previous filtering iterations. It can be replaced by a record, removing spurious memory-based dependences. Furthermore, the work must be distributed over independent workers then merged into a single output stream. To eliminate data copying overhead, the implementation leverages decoupled data access and synchronization, and the extended Kahn semantics where multiple workers deterministically produce data in exclusive index ranges.

Figure 3.11 illustrates the concurrency exposed in `fmradio`. On the left, 6 processes called FFD (Float input, Float output and Double taps) account for most of the computation load. Two of them operate at twice the sampling rate of the two others, involving twice the number of "taps" and twice as many computations. This suggests to balance the load by creating twice as many instances for the heavier ones. The figure dashed box details the data-parallelization of an FFD process, sharing the work into two instances. Figure 3.13 summarizes the speedups achieved with GCC 4.3 and different optimization options. The baseline is the sequential (original) version compiled with `-O2` (no vectorization, less optimizations); it runs in 13.65 *s* on Xeon. These results confirm the scalability of Erbium on a real application. They also confirm its compiler-friendliness, GCC's automatic vectorizer being capable of aggressive loop restructuring in presence of concurrency primitives and view accesses.

Figure 3.12 illustrates the concurrency exposed in `802.11a`. The data-flow graph is more unbalanced than `fmradio` and it is not fork-join. Both `frequency_sync` and `fine_time_sync` processes are stateful, i.e., they need to be decoupled from the rest of the pipeline to enable data-parallelization [60]. Figure 3.13 displays speedup results. The *Combined* column shows the benefit of exploiting both task-level and data parallelism. Strict data parallelization even degrades performance on Xeon due to work-sharing overheads.

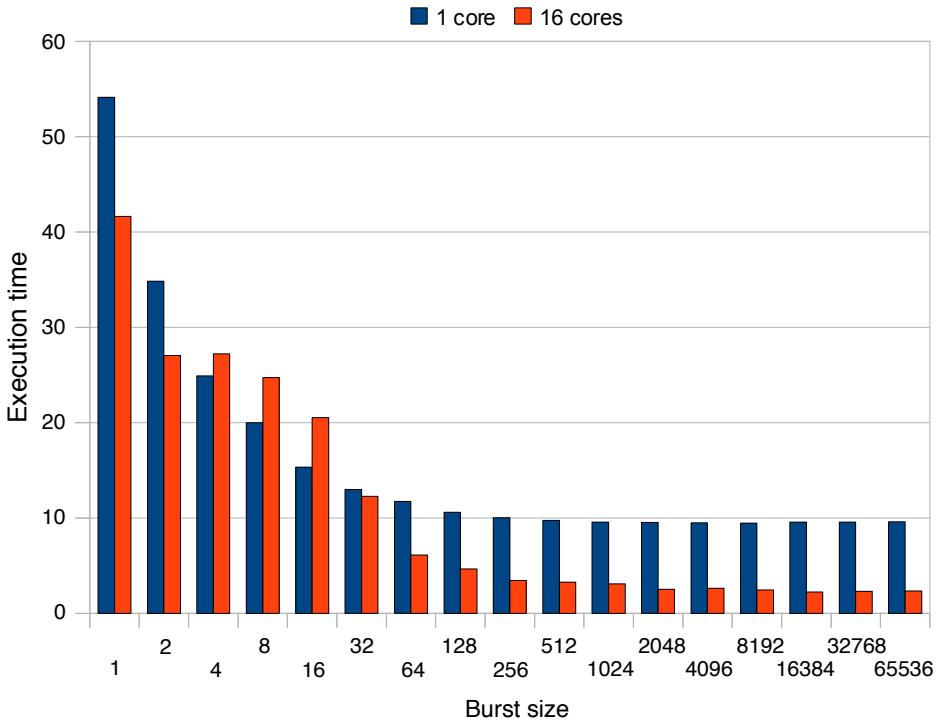


Figure 3.15: Performance of `jpeg` on Opteron.

On `jpeg`, the systematic decomposition of the application exposes 23 computational tasks, communicating by exchanging burst of pixels (or coefficients, depending on the filter/stage). In the uncompressed data stream, 64 pixels/coefficients correspond to 1 macro block. The objective of this experiment is to demonstrate the benefits of Erbium on a real application with extremely fine grain tasks. Most of the fine-grain tasks can be further data-parallelized, but we restrict ourselves to a task-parallel version for the purpose of this experiment. Figure 3.15

shows the results on Opteron, running the decoder once on a 4288×2848 image. The vertical axis is execution time in milliseconds, the horizontal axis is the burst size in pixels. For single-core execution, the performance plateau is achieved for bursts of 128 pixels, or 2 macro blocks. For a 16-core execution, the performance plateau is achieved for bursts of 512 pixels, or 8 macro blocks. Comparing the best 16-core and single-core versions, we achieve a 4.85× speedup on Opteron. Most important, the break-even point (defined as the burst size where 16-core performance outperforms the best single-core performance) is *1 macro block only*. These numbers confirm that Erbium succeeds in exploiting fine-grain thread-parallelism on real applications, although better results could be achieved combining task-level and data parallelism. This is encouraging about the scalability on future manycore architectures, where data parallelism alone does not scale.

The tradeoff between task, pipeline and data-level parallelism depends on the target architecture, and is becoming one of the key challenges when adapting a computational application to a new platform. Our results show that Erbium is an ideal tool to explore this tradeoff. Overall, Erbium leverages much more flexible, scalable and efficient forms of parallelism than restricted models.

3.4.3 Comparison with Lightweight Scheduling

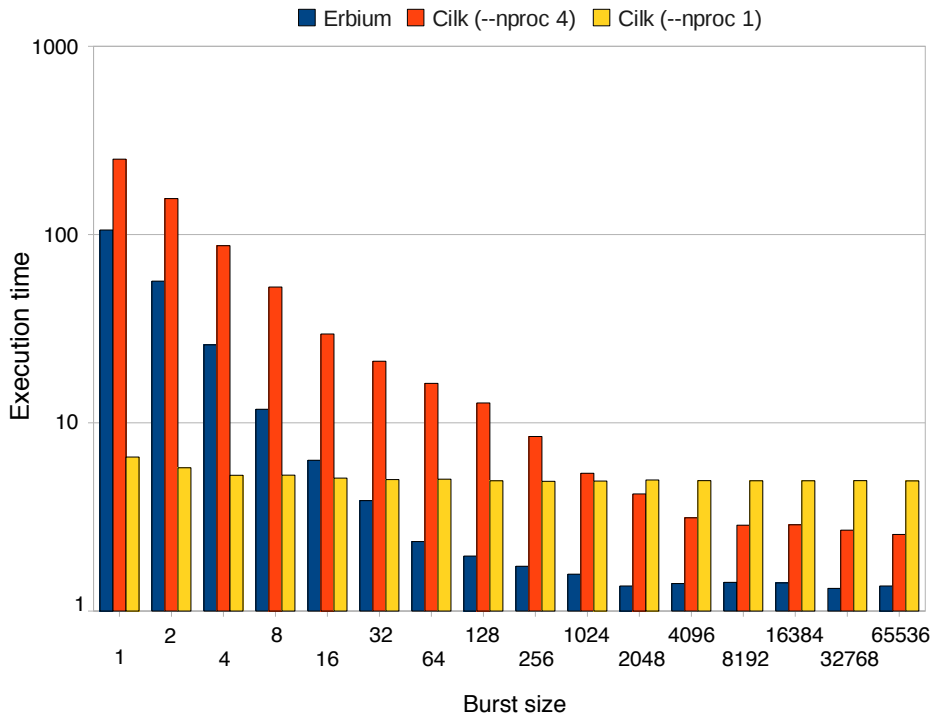


Figure 3.16: Streaming processes vs. short-lived tasks.

Erbium differs from the common parallel runtimes where concurrency is expressed at the level of atomic, short running tasks. Figure 3.16 compares the execution time of the `exploration` synthetic benchmark with a Cilk implementation spawning short-lived user-level tasks [30]. We consider the Core 2 target, and Cilk is run with the `--nproc 4` option to generate parallel code, and with the `--nproc 1` option to specialize the code for sequential

execution. The baseline sequential execution takes almost $7s$ for the finest synchronization grain, and $5s$ for larger ones. The parallel Cilk version with the finest synchronization takes $221.4s$ and the corresponding Erbium version takes $107.7s$. The performance gap widens significantly for intermediate bursts sizes, and reaches almost $5\times$ when the Erbium version reaches its performance plateau. But the most important figure in practice is that the Erbium version breaks even for grain size $80\times$ smaller than Cilk. It demonstrates that communications among long-lived processes are an essential abstraction for scalable concurrency.

These numbers also explain the poor performance of the Cilk FFT implementation. Data-parallelism dominates the scalability of the FFT, and Cilk incurs a noticeable scheduling and synchronization overhead for data-parallel execution. Erbium avoids this overhead by running data-parallel tasks fully independently, and implementing synchronizations across butterfly stages at a much lower cost.

Erbium is also compared with StarSs, in its SMPSs flavor [52]. StarSs is perfectly suited to express our data-flow applications. However, its current execution model relies on lightweight scheduling. Our experiments with `fmradio` show that StarSs achieves $3.88\times$ speedup on Xeon and $2.97\times$ on Opteron, 3 to 4 times less than Erbium.

Short-lived atomic tasks may be better supported with dedicated hardware [48]. This is also the case for streaming communications, as illustrated by the TTL approach [40]. Of course, lightweight threading techniques are still required for load balancing and to increase the reactivity of passive synchronizations (blocking *update*/*stall*).

3.5 Distributed memory implementation

When comparing with shared memory architectures, distributed memory architectures are harder to support targets. Distributed memory architecture imply significant modifications to existing implementation. Example of such modifications are to adapt the synchronization primitives, but as well to deal with explicit communication and user-level scheduling, which are in many cases requirements for this type of architecture.

In the context of this thesis, we developed a simple prototype implementation for the IBM Cell Broadband Engine [42]. Such implementation did not focus its effort on performance but rather on exploring Erbium's support for distributed memory systems. This section does not explains this precise implementation. However, the ideas about to be presented are clearly an evolution of that implementation and the result of a greater understanding of the Erbium language semantics and primitives. Cell, apart from being distributed, is an heterogeneous architecture, making compiler code generation even harder.

There is a very large range of distributed memory machines with very different range of properties and very big distinctions regarding programmability and operation performance. Such architectures most recently have hardware extensions to data communication. As an example, the IBM Cell supports both asynchronous direct memory access and point-to-point message passing in hardware (aka. mailboxes in Cell). Apart from hardware implementations, there are several runtime libraries supporting similar operations, abstracting the programmer from the complexity of such operations. An example is MPI-2 API implementing *send* and *receive* operations based on point-to-point communication. In comparison with Cell mailboxes, MPI-2 has very similar semantics, however it is not limited to single sized 32 bit messages.

Point-to-point messages can be understood as very small one-to-one FIFO buffers con-

necting distinct CPUs/cores. The *send* primitive inserts a message in the buffer while the *receive* primitive collects messages. Each core when calling *receive*, checks for new elements in its respective buffer. Depending on the implementation, both *send* and *receive* block when the buffers are full or empty, respectively. Many architectures or message passing libraries allow *receive* to have either a pooling or interruptible implementation. In a pooling implementation, the application should periodically checks for incoming messages. Interruptible version allows a process to provide a callback function which is executed each time a new message arrives.

Cell implements point-to-point message passing in hardware making it much more efficient when comparing to software solutions such as MPI [3].

Direct memory access (DMA) is another way to transfer data between cores, being most efficient when transferring bigger memory regions. In most cases any CPU in the system can schedule a transfer between any memory region. DMA operations occur asynchronously to every CPU execution and, apart from the requesting CPU, no one else is notified of the operation or even is able to detect pending or terminated operations from or to its local memory. Depending on the DMA engine available, DMA transfers can be possibly executed out-of-order, even when scheduled from the same processor. DMA operations in the Cell architecture can be performed between both "Power Processor Element" (PPE) and the "Synergistic Processing Elements" (SPE), as well as between any two SPEs. As the execution order is not guaranteed, DMA operations cannot be used for inter-processor synchronization. Nevertheless, thanks to its high efficiency when transferring big blocks of memory, it is an excellent candidate to perform buffer data copies.

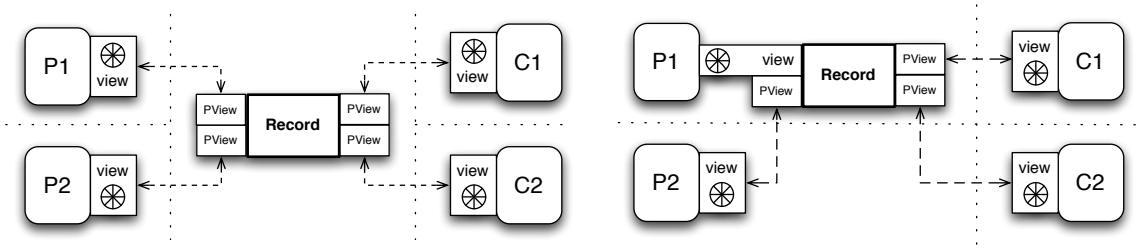
3.5.1 Shared vs. distributed memory models

The X86 libEr implementation makes extreme use of architecture abstractions, exploiting memory and cache properties, such as cache coherence and memory consistency models, minimizing its synchronization and data communication overheads. As such abstractions are not available in distributed memory architectures, implementations have to use explicit low level synchronization calls (synchronous messages) and data transfers.

In a Erbium implementation for distributed memory architecture, record events data must be stored next to the view data structure within the local memory region assigned to each core. Each process contains its private view buffers and data communication occurs explicitly through data transfers between producer and consumer processes.

Each data transfer is a consequence of both *transfer* and the synchronization (*update* and *commit*) primitive calls. The minimum computation is done by either an independent management thread or by one of process threads connected to the record. A management thread has direct access to the record data structure and to a set of replicas of all the views connected to the record. The original view data structures are allocated in the process local memory region. Using the replicas, the management thread is able to decide on lower and upper record indexes, for the now "virtual record buffer"[1], and to perform asynchronous data transfers between the existing view buffers without directly communicating with the original view data structures, allocated in the process local memory region.

[1]Called virtual considering the language semantics. Although the language presents it as allocated next to the record data structure, in a distributed memory implementation, such buffer might not exist but instead be distributed through all of the process local memories.



(a) Centralized manager thread (b) Assigned manager process thread

Figure 3.17: Distributed memory model implementation overview.

Figure 3.17 are two possible scenarios of the distribution of all the previously mentioned data structures in a 2 producers and 2 consumers communication example. Dotted lines represent the boundaries of each of the memory regions where processes are instantiated.

In Figure 3.17(a), the record is allocated in an independent memory region and is used by a single dedicated management thread that does its computation based on the view replicas (PView) directly connected to it. Each of the processes, through its private views, sends a message to the management thread, updating the indexes of the view replicas connected to the record. The management thread, using the view replicas, performs a minimum computation exactly like the shared memory version, but before any message is sent to the dependent processes, the respective data transfers are asynchronously scheduled up to that minimum. As soon as the referred transfers have concluded, the view replicas indexes are updated, and a message is sent to update the process local views.

Figure 3.17(b) is similar to the previous diagram but this time the record data structure is allocated next to one of the processes. With this design, there is no need for an independent management thread. The management is performed by one of the process threads, which performs the minimum computation right after any of its calls to a *commit* or *release* primitive. In such cases, the minimum computation is dependent on the individual progress of the processes and if we assume all the record processes are load-balanced, such minimum computation does unbalance them.

With all the variety of distributed memory architectures, it is impossible to present a single implementation that will suffice to all possible variations.

3.5.2 Process synchronization

Distributed memory architectures synchronization involves the negotiation of different memory regions processes. Point-to-point communication must be used to synchronize view replicas with the original views, distributed through the different memory regions.

Figure 3.18 shows a sequence diagram of the existing interaction between a producer and a consumer processes in a distributed memory environment. Both the producer and consumer communicate with a record management thread through point-to-point messaging.

In the example, the producer process has already written data in its private local view buffer. During *commit*, its local view index is updated and a message is sent to the record management thread. The management thread when checking for new messages realizes the producer did *commit* and upgrades its referred view replica with the message passed index.

After checking for messages and upgrading view replicas, the management thread can

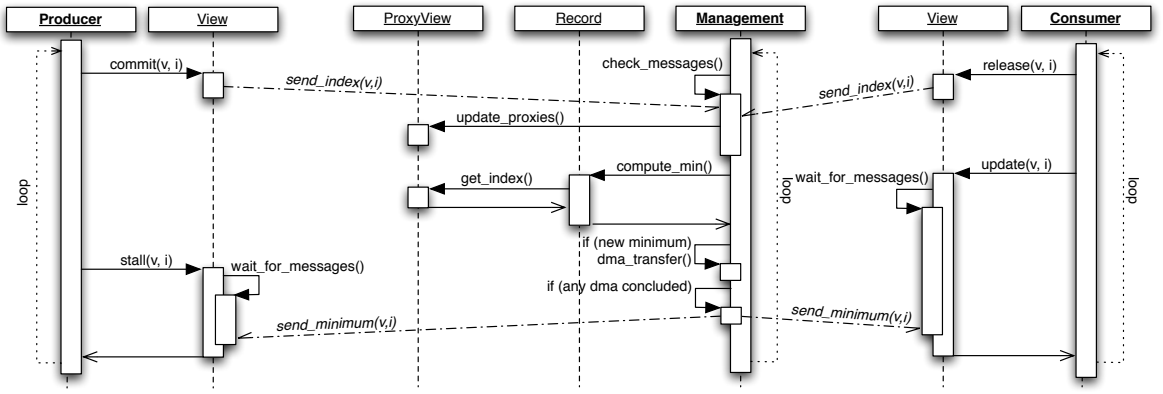


Figure 3.18: Synchronization related communication.

calculate both lower and upper record indexes (minimum computation) computing the bounds of the "virtual shared buffer". This computation is equal to the shared memory version and allows the record manager to identify which events can be removed from producer buffers and which events became available and are transferable to consumers. The necessary data transfers are now done through asynchronous DMAs as is explained in the next section.

When DMAs conclude, the record management upgrades the replicas indexes, as well as send back a message to both producers and consumers specifying the new view indexes, used to unblock both stall and update calls, respectively.

Although not represented in the diagram, *release* has a similar implementation as *commit* and forces DMA transfers not to be initiated if not enough space is available in the consumer view local buffer.

3.5.3 Data communication

As explained in Chapter 2, synchronization in Erbium is the engine for deterministic data communication, which, although expensive, can be minimized in most cases, by explicitly tuning its granularity with respect to data communication. Furthermore, data communication is predicted based on the index based synchronization, allowing anticipation of data transfers ahead of its usage.

Multiple buffering techniques together with prediction of data usage and a well load balanced application allow the runtime support to hide memory latency reducing execution time. Mostly in multiple producers or multiple consumer examples, either by requirement or to achieve optimal results, processes might benefit from partial data copies using the *transfer* primitive.

Although the *transfer* primitive has no relation to the language determinism in shared memory, it is crucial in distributed memory architectures. While producers directly write into a shared buffer in a shared memory implementation, in distributed memory multiple producers will write into private independent buffers and only later each of these private buffers are merged. In order to merge both the local view buffers into a unified ,"virtual" or not, record buffer, it is essential to identify the relevant indexes from each of the record producers. Consumer processes also benefit from *transfer* primitive, more precisely in work-splitting processes.

The combination of both producers and consumer *transfer* calls allows to identify the table of data transfers required for the application progress.

In this distributed memory approach, the *transfer* primitive sends a point-to-point message to the record management thread. Once identified, such message should be stored within the respective view replica. Data sent by *transfer* stored in writer replicas (aka. producer transfers) announce a range of elements which will become available through that view, while reader replica *transfers* (aka. consumer transfers) define the elements that will be used through that view.

Data communications (DMAs) occur right after all the producer processes have committed. The record management thread collects all the *commit* and *release* messages and computes the lower and upper indexes. By traversing all the producer and consumer *transfer* call logs and intersecting their provided argument ranges, it is possible to identify the necessary DMA transfers and the origin and destination for such transfer. Apart from intersecting producer and consumer *transfer* primitive calls, it is necessary to take synchronization into consideration in order to avoid such DMAs to overlap consumer buffer positions still in use. Furthermore, the DMAs should also be taken in consideration by synchronization primitives to, for example, allow producer buffers re-usability once all the necessary DMAs have concluded.

Each DMA transfer is associated with the pair of the PViews (origin and destination view replicas). The record management thread, while frequently iterating on all views, verifies if any of its executed DMAs has terminated, in which case, the view replicas are updated with the minimum that generated such DMA transfer and a message is dispatched to the associated view processes, as briefly explained in Figure 3.18.

Algorithm 18 details data communication presented in previous paragraphs, as two pseudo code functions previously used in Figure 3.18. The function *perform_transfers* traverses both the producer and consumer *transfers*, correlating them and initiating the necessary DMA copies. The *check_terminated_dmas* function verifies, per view, if all the previous scheduled DMAs have terminated, updating the view replicas, and messaging the respective process views.

In more detail, *perform_transfers* correlates the producer related transfers to the consumer ones, by intersecting the ranges of both the *transfer* calls (Lines 2 and 3). Moreover, only the events with index below the current reader view latest release or record *upper_index* should be considered for DMA (Line 5). If from the above verifications there is any event data that should be copied, a DMA transfer is executed and the *id* of such execution is stored within both the origin and destination view replicas, associated with the current record upper index (Line 8).

The function *check_terminated_dmas* traverses all the view replicas looking for terminated DMA transfers, using the previously stored DMA *ids* (Line 19). Once the DMAs are executed, the consumer *transfer* ranges are updated by removing the finished DMA transfer ranges from the list of pending transfers (Line 10).

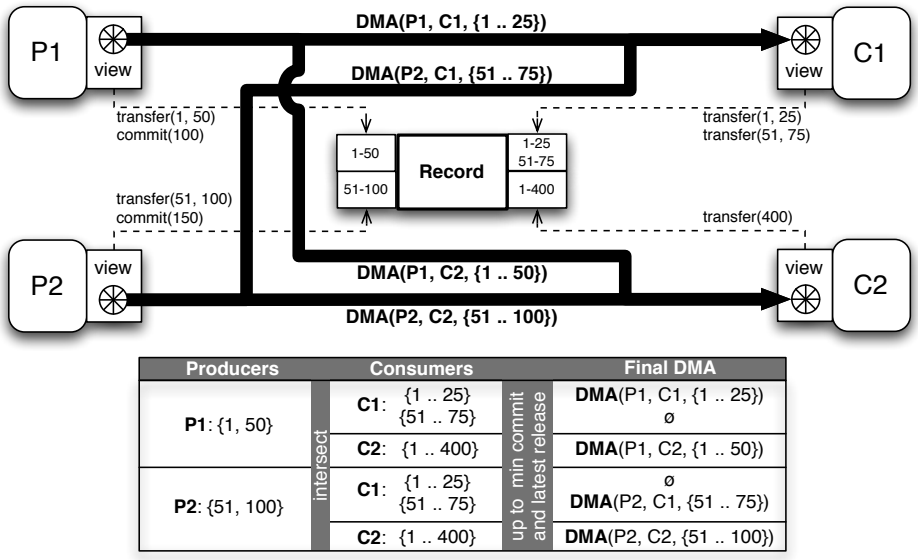
Figure 3.19 is an example record and view replicas state and the execution outcome of function *perform_transfers*.

The presented design abstracts the need for a record buffer, allowing a direct DMA transfer from the producers to the consumers memory space directly. Nevertheless, it is necessary to perform a pre-negotiation of all the scheduled DMA transfers. In this prototype, the negotiation occurs by correlating all the *transfer* primitive calls, previously executed by

Algorithm 18 DMA transfers based on *transfer* primitive calls and minimum index integration

```
 1: function PERFORM_TRANSFERS(record)
 2:     for wt ∈ record.writers.mapToTransfers() do
 3:         for rt ∈ record.readers.mapToTransfers() do
 4:             max_index ← MIN(record.upper_index, rt.pview.index)
 5:             index_set ← wt ∩ rt ∩ {0.. max_index}
 6:             if set ≠ ∅ then
 7:                 dma_id ← DMA(wt.view, rt.view, index_set)
 8:                 wt.pview.hash_dmas[record.upper_index].push(dma_id)        ▷ Register DMA
 9:                 rt.pview.hash_dmas[record.upper_index].push(dma_id)
10:                 rt ← rt − (index_set)
11:             end if
12:         end for
13:     end for
14:     remove_all_wts_below(record.lower_index)        ▷ Remove no longer used writer transfers
15: end function

16: function CHECK_TERMINATED_DMAS(record)
17:     for pview ∈ record.writers ∪ record.readers do
18:         for min_index ∈ pview.hash_dmas.keys().order() do
19:             if ∀ dma_ids ∈ pview.hash_dmas[min_index] : dma_ids have terminated then
20:                 pview.record_index ← min_index
21:                 send_message(pview.processor, min_index)
22:             end if
23:         end for
24:     end for
25: end function
```



Producers		Consumers		Final DMA
P1: {1, 50}	intersect	**C1**: {1 .. 25} {51 .. 75}	up to min commit and latest release	**DMA**(P1, C1, {1 .. 25}) ø
		C2: {1 .. 400}		**DMA**(P1, C2, {1 .. 50})
P2: {51, 100}		**C1**: {1 .. 25} {51 .. 75}		ø **DMA**(P2, C1, {51 .. 75})
		C2: {1 .. 400}		**DMA**(P2, C2, {51 .. 100})

Figure 3.19: Diagram representing the view buffer DMA transfers based on Algorithm 18. Dashed arrows represent the point-to-point communication by each of the concurrent processes. Wide arrows represent DMA transfers. The table underneath is a graphical representation of the *perform_transfers* algorithm, presenting both producer and consumer transfer ranges and its outcome DMA data transfer. The two left columns represent the traversal through all the producer and consumer *transfers*. The last column is the result of the correlation between the producer and consumer *transfer* primitives.

connecting views, enforcing that all these messages are collected by the record view replicas.

Collecting *transfer* primitive "meta-data" might not always be possible or efficient. An other possible solution is to permit the record data structure to contain its own private buffer or even mark one of the view buffers as the master one. In this case, instead of collecting the *transfer* "meta-data" within the view replicas, the processes directly schedule DMA transfers after each *commit*. Symmetrically to *commit*, *update* also initiates a transfer from the master buffer into its local view buffer, as soon as it receives a new minimum message (Figure 3.18).

In an application with many producers and consumers, concentrating so many DMA transfers into a single memory region can saturate its core local memory bandwidth, producing severe slowdowns when comparing to "meta-data" storing and processing cost.

As mentioned before, there are hundreds of properties distinguishing each distributed memory machine, making it impossible to produce a single optimum implementation, or to cover all the possible implementations. This design is a prototype implementation close enough to the latest generation of distributed memory architectures properties. The presented approach was not implemented and is not intended as a contribution to distributed memory runtime support research topics. Presented work is a speculative Erbium implementation to distributed memory architectures. Efficient Erbium runtime solutions for distributed memory are open for improvements and future research.

3.6 Summary

This chapter presented libEr, a x86 busy-waiting version of the Erbium runtime support library. The goal was to generate the synchronization primitives with the lowest possible overhead, regardless of scheduling algorithms or scheduling policies. The x86 architecture with its TSO memory consistency model allowed us to design an implementation free of memory barriers or atomic operations.

Lazy implementations[1]were also discussed, not necessarily providing an implementation but rather a detailed description of its primitives/data-structures implications. When comparing with busy-waiting, we made suggestions regarding user-level threading and scheduling approaches.

The last section of the chapter presented a possible implementation of an Erbium runtime support for distributed memory architectures, explaining the main differences between shared memory and distributed memory architectures, and the necessary data structures and algorithmic changes to adapt the shared memory implementation to distributed memory architectures.

During the chapter, a short but precise set of experiments are presented, benchmarking and comparing libEr with other language runtimes and language-specific properties, such as short living vs long living processes. We evaluate both a synthetic benchmark and real applications, not only proving the efficiency of libEr but also Erbium's ability to support a variety of streaming real world applications.

The next chapters further detail the integration of the Erbium language as a compiler intermediate representation, avoiding obfuscating its optimizations. We also show Erbium's suitability to implement parallelism, enhancing optimizations for Erbium and we extend the original compiler optimizations, further improving its primitives.

[1]Lazy implementation refers to any implementation relying solely in resourceless waiting primitives. In other words, blocking operations do not contain active loops as a way to block further execution, eventually spending CPU execution time.

Chapter 4

Compiler Design

Compilers are considered by many the most fundamental software components developed to date. Nowadays, every programmer relies on,at least one compiler framework in its work-flow on a daily basis and, as hardware complexity increases, one cannot predict that such scenario will ever change. Most advanced compilers tend to be extremely complex and frightening. Nevertheless, such complexity is the outcome of decades of very careful design evolution, while preserving its generated code credibility and correctness.

As the number of parallel architectures increases, so do the attempts to create multi-purpose higher level languages. Historically, many high-level languages avoid mainstream compiler integration by being implementing their own source-to-source compilers[1]. Such compilers convert higher level parallel languages into a known sequential language, combined with some external runtime library calls, later compiled by a mainstream sequential compiler.

Such approach allows the language designer to quickly create a compiler for its language, without having to deal with the complexity of modifying a mainstream compiler. Moreover, such compilers exploit the language and target architecture properties in order to optimize its generated code. However, source-to-source compilers also introduce code complexity, tending to generate obfuscated code and eventually disable many of mainstream compiler optimizations.

In order to minimize its penalty, source-to-source compilers tend to duplicate the most fundamental code analysis and sequential optimizations already available in the mainstream compilers, as an attempt to improve the quality of its generated code.

The Erbium language provides the means to express stream-like parallelism and its integration within mainstream compiler without obfuscating existing compiler optimizations. When correctly integrated, Erbium applications are able to execute traditional optimizations as the sequential counter part application would, as well as to perform some finer local to process parallel optimizations, possibly enabling other sequential and parallel optimizations.

Most higher-level parallel streaming languages use abstractions such as graphs[2] in order to represent and combine data-communication and synchronization. Such languages, while limiting the language semantics, provide very good initial parallel code information such as process dependencies. An example is StreamIT (SDF programming model), well-known for its static scheduling and optimizations, as exploited by Gordon et al. [34]. When compared

[1]Source-to-source compiler is a type of compiler that takes the source code of a programming language as its input and outputs the source code into another programming language.

[2]Task graphs are many time expressed as pragmas [17], or even XML files

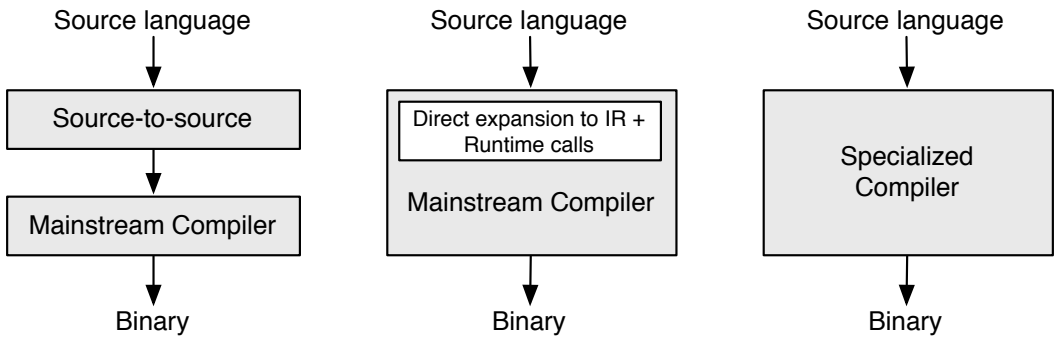


Figure 4.1: Three most common parallel language compilation schemes.

to Erbium, such languages have more limited and simpler semantics, as illustrated by their static task activations and task graphs. Moreover, parallelism properties are deeply integrated in the language semantics. For example, languages combining synchronization and data communication, are not able to easily adjust their synchronization granularity.

Higher level languages provide easier means to obtain a clearer static information of the different process dependencies and relations. On the other hand, the Erbium language has different independent primitives to deal with communication, synchronization and process creation, making it a very expressive low level language. Such expressiveness comes with the cost of unpredictability, making it very hard to deduce process communications and its properties.

As an example, in the specific case of a fusion of two processes, it is necessary to predict the number of activations of each of the fusing processes to decide on the transformation validity. This information is not easily obtainable from the Erbium representation, a price to pay for its expressiveness. However, languages based on SDF and CSDF computational model provide sufficient information allowing to easily it, either at its front-end lowering to Erbium.

Erbium high expressiveness and its close to hardware semantics, allow the compilers to represent parallelism, while maintaining the sequential language properties and preserving the original mainstream compiler sequential optimizations. Moreover, Erbium allows lower level properties such as synchronization granularity or data-communication to be further optimized.

4.1 Current generation parallel compilers

Most higher level parallel languages lack a good compilation environment, capable of both exploiting its language properties and supporting traditional optimizations.

Mainstream compilers focus on optimizing sequential code, containing no or very few optimizations for parallel code. At best, they try to leverage the weak memory models of the source language, as demonstrated by Boehm and Adve in [18], while preserving sequential code optimizations in the presence of shared memory multi-threading.

Figure 4.1 presents some of the typical strategies used for parallel compilation approaches.

The left-hand side diagram, presents a source-to-source compiler that is responsible to optimize and translate the parallel language constructs into some well-know sequential language, usually C, patched with function calls to the language runtime support library. An

example of a source-to-source compiled language is StreamIt.

The second diagram (middle) goes a little deeper and implements the exact same translation inside a mainstream compiler. However, such translation occurs very early in the compilation flow, introducing similar runtime calls, resulting in the exact same drawbacks as previous approach. This is the case of the OpenMP language extension in GCC.

The last approach (right-hand side) is used by high-level synthesis tools (e.g. Catapult C, Synopsys-Synfora, Bluespec, etc.) where a totally new compiler is defined from the ground-up taking into consideration the language properties. This approach has not been as popular for parallel programming. Although optimal, such approach implies very high development efforts considering how complex developing a compiler can be. Moreover, such compilers tend to be very restrictive for the specific language as well as target architectures.

Source-to-source compilers require typically the use of an external runtime library, implementing all the language primitives not available within the target language. When presented with external runtime library function calls, mainstream compilers are not able to access those external function definitions and so are unable to optimize such calls or any of its surrounding code.

Inlining such library calls within the generated code produces very little performance improvement. Although all of the runtime library code is available, it is very likely that the inlined code will contain very low level synchronization function calls, such as atomic operations or volatile shared variables, creating similar side effects as the external library function calls.

In order to achieve performance, source-to-source compilers tend to replicate most of the mainstream compiler sequential optimizations obtaining reasonable performance improvements from its generated code.

4.1.1 Redesigning mainstream compilers

Mainstream compilers must be able to represent parallelism in thier intermediate representations. Such intermediate representation should be designed to express the properties of a wide range of languages, while preserving well defined semantics and dependencies for all the language primitives. Moreover, mainstream compiler implementations must be aware of such parallel primitives and not only be able to optimize those, but also its surrounding code.

Higher level parallel languages must generate code using the newly defined compiler primitives. Any possible higher-level optimizations should be executed before code lowering. Code generation can occur either internally in the mainstream compiler or through an external source-to-source compiler, performing optimizations and outputting intermediate language compatible code. Figure 4.2 presents these two possible compilation scenarios.

By supporting higher language properties, mainstream compilers are able to perform a better task preserving the original language semantics while optimizing its code. Nevertheless, higher level languages should be able to lower to the newly intermediate language, abandoning its runtime library support and adopting the intermediate language runtime support (libEr for example). Erbium is such intermediate language.

Erbium language primitives provide deterministic parallel synchronization and data communication. Moreover, the primitives have very clean dependencies, allowing its integration in mainstream compilers while preserving legacy optimizations compatibility and usability.

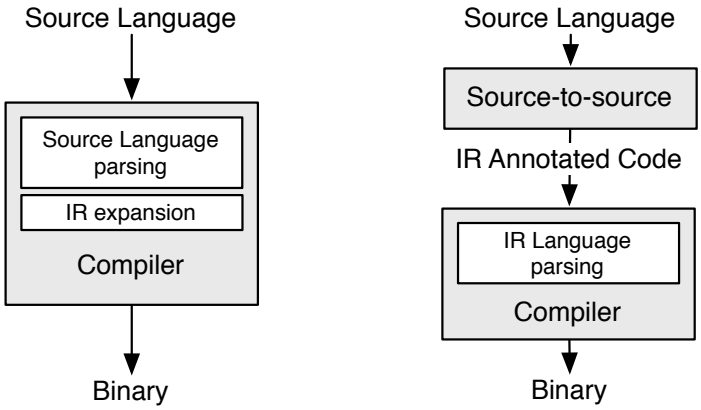


Figure 4.2: Possible compilation schemes using a parallel intermediate representation.

4.2 Mainstream compilers - GCC

Compilers are one of the most comple software achievements since the beginning of computing era. One example is GCC containing hundreds of thousands lines of code and which has evolved into a software "dragon" as compilers are typically represented [7].

This frightening effect of GCC tends to repel developers and companies from its adoption. The LLVM compiler infrastructure is the most recent GCC competitor, clearly winning with respect to code complexity and learning curve. Although, GCC is still a clear winner in respect to number of supported languages and targets, along with generated code performance.

GCC is separated in three independent compilation stages:

- front-end, where all the parsing is performed besides the lowering of the code to its intermediate representation (GIMPLE),
- middle-end, where most of the code analysis and optimizations occur, involves code conversions between several representation forms, as well as the definition of call and control-flow graph data structures,
- back-end, converting previous representation closer to assembly level, yet possible to be optimized.

Figure 4.3 represents a global view of the GCC compilation flow, detailing more into the structure that composes the compilation environment. The middle-end stage is further split into inter-procedural analysis (IPA) and optimizations passes.

These collections are composed of many passes sequentially executed through the compilation flow. The pass execution ordering defines the compilation flow, as well as the available data structures and representations available at a specific compilation stage.

IPA passes are executed once per compiler execution. Examples are OpenMP pragma lowering, call-graph and static single assignment (SSA) generation passes.

As specified in Figure 4.3, passes like vectorization, partial redundancy and dead-code elimination are defined as optimization passes. Optimization passes are executed multiple times per compilation and at least once per compiling function. Each pass, apart from its

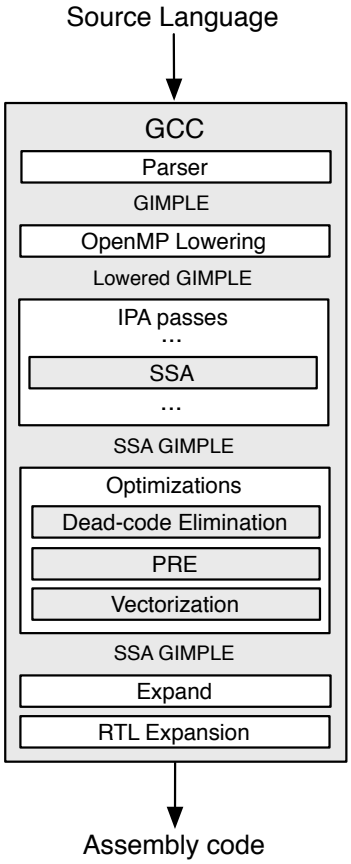


Figure 4.3: GCC internal structure, separated into its three compiling phases, intermediate representations and the most relevant optimizations.

own code, that performs its predefined task, is associated with:

- a *gate* function checks if the pass should execute. It is used to check for compilation generated properties or even, the most common case, if the pass was selected for execution through compilation arguments,
- *provide* and *destroy* properties, defining whenever some intermediate representation or data structures are created or destroyed by the pass. For example, control-flow graphs, call graph, or the SSA form,
- *todo* lists, providing the means to enforce code executions before and after the pass execution, generally used as a preparation for the pass or reconstruction of broken datastrucures. An example is requesting for recomputing the SSA form or even recompute the control-flow-graph, after code transformations.

Passes can also be structured hierarchically where execution of the child pass requires the previous execution of the parent. An example of such pass hierarchy is the auto parallelization passes (vectorizer), where the parent pass identifies and prepares the vectorizable loops, and the child passes perform the actual code transformation.

The GCC middle-end intermediate representation (named GIMPLE) is a three-address representation [7] where all the control statements are converted into conditional branches

(if statements and gotos) commonly known as spaghetti code [7]. After the lowering stage execution, all of the middle-end passes read and generate GIMPLE code. In the front-end, GIMPLE is generated either directly after code parsing (in case of C and C++ source-code) or through a more abstract higher level GENERIC language, later also converted to GIMPLE.

During IPA passes, GIMPLE is converted into SSA form which is preserved almost until the end of the middle-end stage.

The **call graph** is a directed graph representing all the compilation unit function calls. Nodes represent function definitions and edges represent function calls. Each node contains a direct reference to the GIMPLE node representing its function definition. Edges contain a reference to the precise statement that generated the edge and connects both the nodes from the respective caller (function containing the call statement) and callee (function being called).

The call-graph data structure is generated right after the front-end execution and is the main driver for the remaining compilation stages. For example, it is the data structure used by the pass manager to determine the order in which every function should be compiled.

Throughout the compilation, each pass can access this data structure to collect caller and callee information, in addition to store any meta information generated and used by later executing passes. Examples are the profiling passes, collecting expected number of executions of each function and boolean flags marking the existence of valid control-flow and alias analysis data structures. Many optimizing passes access such information in order to decide to apply a particular optimization. One clear example is the function inlining pass which uses, in its decision heuristic, the execution counter collected during the profiling pass to decide the set of function calls to inline.

The **control-flow graph** (CFG) for a specific function is generated during the lowering stage and is widely used through all the compiler passes. A control-flow graph is composed of basic block nodes, which are regions of code containing no control flow statements such as jumps or jump targets (labels). Each basic block contains a list of statements whose first one should be a label statement and the last one a jump statement. The CFG edges represent basic blocks connections based on the control-flow of the compiling function, more precisely on the last statement of the source basic block.

Apart from control-flow information, the CFG edges also store detailed information such as distinguishing conditional jumps (false or true jump edges), fall-through jumps (no conditional) or even edges resulting from exception handling jumps. Like the call-graph data structure, profiling information is stored within the nodes of the CFG. Later during compilation, the CFG is extended with loop nest (loop tree) information, which is the underlying data structure used by scalar evolution analysis and loop optimization passes, such as vectorization. Scalar evolution analysis were studied and implemented in GCC by Pop and published by Pop et al. in [70, 71]. Scalar evolution analysis are used later in Chapter 6 in the context of Erbium optimizations.

GIMPLE SSA

The SSA form [24] is also a three address code (GIMPLE) representation, although, as the name suggests, each variable is assigned only once. For every variable assignment, a new variable is created with same name postfixed and a unique version number. Every following variable access is substituted by the new defined version.

```
1  int i = 0
2  while(i < 100) {
3    do_something()
4    i += 1;
5  }
```

Original Code

```
1  BEGIN:
2    i = 0
3  LOOP:
4    if(i >= 100)
5      goto END;
6  LOOP_BODY:
7    do_something();
8    i = i + 1;
9    goto LOOP_END;
10 LOOP_END:
```

GIMPLE

```
1  BEGIN:
2    i.1 = 0
3  LOOP:
4    i.2 = phi < i.1, i.3 >
5    if(i.2 >= 100)
6      goto END;
7  LOOP_BODY:
8    do_something();
9    i.3 = i.2 + 1;
10   goto LOOP_END;
11 LOOP_END:
```

GIMPLE (SSA)

Figure 4.4: Simple GIMPLE and SSA loop conversion.

Nevertheless, it is not always possible to predict the exact assignment expression for a specific SSA variable, for example when multiple definitions of a single variable can reach a single usage statement or, in more detail, when two control-flow paths merge together while both paths define the same variable. In this case, SSA conversion introduces a new variable definition, a ϕ-function holding the different sources to be merged as arguments. This abstraction provides the necessary means to disambiguate different reaching definitions.

Figure 4.4, shows a simple loop code extraction containing the original code (left), the GIMPLE representation generated from the original code (middle), and its SSA form version (right). Notice the ϕ (*phi*) function assigned to $i.2$ specifying the possibility of its value to become either $i.1$ or $i.3$.

By creating new variables for each variable assignment, the SSA form eliminates code code dependences and simplifies use-def chains (the data structure that maps each variable usage to its related definition) by reducing its complexity and generation cost, considering that now every variable usage has a single definition. The SSA form by default also performs live range splitting, or in other words, distinguish and split unrelated uses of the same variable. A simple example is the split of a variable that in used as induction variable in two unrelated loops. After live range splitting, such variable has a greater chance to get promoted into a register.

Apart from this simple optimization, SSA form is, by far, the most relevant representation in mainstream compilers and simplifies optimizations such as:

- constant propagation,
- sparse conditional constant propagation,
- dead code elimination,
- global value numbering,
- partial redundancy elimination,
- strength reduction,
- register allocation.

Near the end of middle-end compilation phase, SSA form is converted back into non-SSA GIMPLE. At the last stage of the middle-end, the GIMPLE representation is converted into the Register Transfer Language (RTL), a representation closer to assembly language used for optimizations such as instruction selection and register allocation.

Alias analysis

Alias analysis provide compilers with the means to disambiguate possible alias variables. In other words, it gives a men to understand when two pointers could possibly point to the same memory region. When two variables point to the same memory region, such variables can create side effects to one another, making value prediction harder, obfuscating compiler analysis and optimizations.

The most simple form of aliasing occurs through immediate assignment of two pointer variables, although this is not the only case. One example is a function call to which the program passes a pointer argument. Unless the compiler can precisely analyze the function code, such function could possibly create a copy of the pointer and eventually later manipulate its data in undetermined ways. In such cases, the analyzed pointer is unpredictable and is considered as "escaped", or in other words, impossible to analyze. Pointers can also be considered to escape, if they are stored as global or stored in other already escaped data structures.

In order to help alias analysis to collect a more precise alias information, the most common and well-known functions have precise alias analysis implementation, rather than escaping any of its pointer arguments. Examples of such functions are standard string operations and allocation functions such as *strcpy* and *malloc*. Let us use the following *malloc* and *strcpy* statements as examples:

```
ptr = malloc(SIZE)      strcpy(str2, str1)
```

Malloc allocates $SIZE$ new bytes in heap memory region. As anyone knows, no *malloc* function, a priory, shares the memory region with any other code that eventually would manipulate the allocated memory. Nevertheless, without a precise alias analysis, the returned pointer from *malloc* is immediately considered escaped.

Strcpy would as well escape both pointers provided as arguments.

GCC solves such alias inconsistencies by supporting "set constraints" which are "a way of modeling program analysis problems that involve sets"[1][39, 62]. It consist of a constraint language to describe operations and the impact of such operations on related statement variables. By applying a set of rules to the constraint sets, it is possible to derive all the possible facts from the variable, resulting in points-to sets for all the involved variables.

Both *malloc* and *strcpy* are defined within GCC as builtin functions. Moreover, each of those builtins is specifically defined with its set of constraints, allowing alias analysis to better understand what happens to its pointer argument variables. More precisely it specifies that none of the functions lets its arguments escape and in case of *strcpy* that the memory pointed by its second argument is copied into the first argument memory.

Set to constraints are further detailed later in the chapter in the context of the de-obfuscation of Erbium primitives.

Built-in functions

In order to better support common operating system functions, GCC provides the means to define prototype function definitions, named built-in functions.

[1]Taken from GCC's code comments

Three different types of builtin functions can be defined in GCC:

- Functions external to compilation-unit whose behaviour is well understood and that the compiler optimizations take into consideration and perform at its best. Examples of such builtins are stdlib, string, libmath functions.
- Functions implemented through special architecture instructions. Examples are the atomic operations available in X86 architectures.
- Function calls introduced through code generation. Although such functions do not exist in the original code, GCC validates their existence and in the end links them to the specific target library. This is the case of the GCC support for OpenMP which from the OpenMP pragmas, generates libgomp function calls (defined as builtins).

Each builtin is defined using some macro pseudo language and, among other things, is able to specify the function name, type of arguments and a set of attributes.

Attributes allow to specify with more detail other properties of a specific function or function call, allowing the compiler to better decide how to apply optimizations, minimizing the cost of those builtin calls and surrounding code. Attributes resemble regular function call attributes, available through the *__attribute__* keyword in many compilers.

Apart from the decision based on attributes, some optimizations rely on the builtin node itself to identify the specific call and make a better job. One example is given by the previous *malloc* and *strcpy* functions, where alias analysis directly identifies these builtins.

Many builtins are directly converted within the compiling application, avoiding the function call overhead and even sometimes completely replaced by compile time computations, removing completely its cost. Examples are the pure mathematical functions (such as cosine). When present in the code with constant arguments, its result is computed at compile time and directly converted by the compiler to its known result (for example, $cos(90)$ is converted to 0 and the original call is removed).

Builtin attributes

In order to allow optimizations similar to the previous *cos* example, it is necessary to precisely type each individual builtin with its most optimistic, yet correct, attribute.

Setting the builtin attributes allows to specify some properties of the specific builtin function, minimizing optimization penalty for such call. The most relevant and common function attributes are:

- pure, specifying that the function has no hidden side effects[1], except from its return value which is computed based only on arguments and/or global variables,
- const, is just like a pure function, although the return value of such function is only a consequence of its arguments and never of any global memory variables,
- hot, informs the compiler that the function is a hot spot of the program, hints for inlining and code locality optimizations.

[1] A function or expression is considering as having side effects if modifies some program state or has an observable interaction with external functions. Moreover, with side effects an application depends on the execution ordering of its side effect statements.

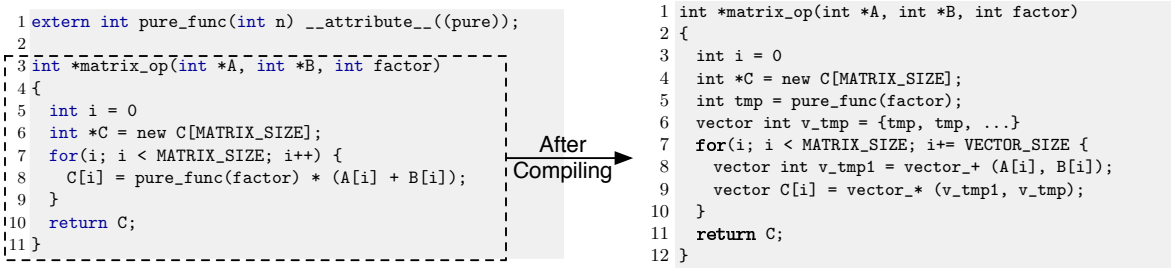


Figure 4.5: Pseudo code demonstrating the compiler impact of a pure function attribute.

Pure or const attributed functions are subject to optimizations such as common subexpression elimination, or loop optimizations. As an example, as such a function only impacts the return value, multiple similar calls to the function can be substituted by a single execution into a temporary variable (later in final binary it can possibly be a single register).

Figure 4.5 demonstrates the relevance of correctly attributed functions. As one can realize, *pure_func* can be considered as a loop invariant call, if it is assumed that such function has no state, or in other words its return value is only dependent on its provided argument. Such functions should be defined as *pure*. The function *pure_func* without its attribute *pure* would be considered as containing side effects, thus the compiler could not consider it as loop invariant. As soon as the function is attributed as *pure*, the compiler could predict that the function call is a loop invariant and consequently hoist it. After this loop simplification, the vectorization pass detects the opportunity and vectorizes the loop. Although the *pure* attribute only enabled *pure_func* to be hoisted from the loop, other optimizations could be applied thanks to this simple optimization, greatly improving code efficiency.

4.2.1 Optimizations

GCC in its current state contains hundreds of passes and each of those either performs analysis or contributes to code optimizations. Examples of optimizations are dead-code elimination, partial redundancy elimination, vectorization and the more recent polyhedral optimizations implemented through the Graphite representation [72].

Partial redundancy elimination (PRE)

Partial redundancy elimination (PRE) addresses, apart from PRE optimization itself, multiple types of code motion, such as loop invariant code motion (hoisting / scalar promotion) and global common subexpression elimination.

PRE works through iterative data analysis predicting where expressions are computed and anticipating new locations where such expressions can be moved, in order to reduce the total number of similar computations executed.

Such analysis implies the computation of availability and anticipability sets, defining where expressions are available (up-safe) or anticipable (down-safe) for all basic blocks of the compiled functions. In a very safe implementation, only operations both anticipable and available at the same program point would be moved. Most compilers PRE heuristics are more flexible and verify for some more special cases where code motion is also possible.

Many times, both availability and anticipability sets are “contaminated” with statements that the compiler predicts as containing side effects, such as operating system locking system

calls, alias escaped memory pointer dependencies or even global variable accesses. Statements with side effects limit the PRE analysis, reducing code motion opportunities.

GCC defines side effect free functions by setting the function attributes as *pure* or *const*. In other words, any function call not marked as *pure* or *const* is assumed as containing side effects, and contaminates availability and anticipability sets, disabling code motion across it.

PRE is further studied in Chapter 6 where it is extended to support Erbium intermediate representation synchronization and communication primitives.

Dead-code Elimination (DCE)

As mentioned before, compilers are implemented in a non destructive manner where each optimization executes only if specific conditions are verified. Moreover, many optimizations are inter-dependent, i.e., are only executed after a successful execution of a predecessor optimization. Dead-code elimination is one of such optimizations. Many programmers judge dead-code elimination as the worthless optimization, when considering their own proficient programming skills. However, DCE is one of the most important optimizations within any mainstream compilation framework due to the compiler optimizations themselves. For example, SSA code conversion creates many temporary redundant variables, later removed by DCE.

As dead-code has no semantical meaning, keeping it can be considered as very costly with respect to its uselessness. Without a dedicated DCE pass, optimizations require to explicitly perform dead-code elimination, simplifying its code analysis.

Vectorization

The vectorization pass in GCC, although highly sophisticated, is applied like a pattern matching algorithm over loops and their data dependencies. Very precise code patterns are necessary in order for vectorization to execute.

A single statement inside a loop can make the difference between a vectorized loop or an aborted transformation. Most of the times, vectorization is unable to execute considering the existence of side effects expressions or functions within the attempted vectorizing loop. Moreover, in order to execute vectorization, it is necessary to move loop invariant expressions or remove the dead-code out of the loop.

The Vectorizer pass is an example of the benefits of under-evaluated optimizations such as DCE, as well as the importance of optimizations interoperability.

Compiler optimizations have very sensible constraints. Whenever the compiler cannot assure a correct code transformation out of some code analysis, such optimization are, by default, ignored and not executed. Such approach to compilation makes sure that the semantics of the compiled application has not changed. A small code detail can make the difference between an optimized and an unoptimized application. Optimization decision heuristic algorithms are one of the biggest challenges to any compiler developer. Moreover, such a complexity is the main reason for the huge repositories of test cases always accompanying mainstream compilers.

Some optimizations are simply impossible to apply considering the lack of details either from the application semantics, or even from its programmer.

Nevertheless, it is not always the programmer fault, but rather the inability to represent

```
void * __builtin_er_alloc_process_instance  (void *process, int  arg_struct_size );
void  __builtin_er_run  (void * process_instance );
void * __builtin_er_wait_for_processes_end  (void);
```

Figure 4.6: Process instantiation buitins.

the code meaning. An example are source-to-source compilers of higher level parallel languages when they are not able to expose the language primitives semantical meaning to the mainstream compilers. As such primitives do not have extreme properties as *pure* or *const* functions, but rather much finer ones, those are impossible to express in current generation of mainstream compilers.

Applications developed in parallel languages and converted into runtime libraries tend to be not as optimizable as the sequential counter part applications. This is the case because of the complexity introduced by the function calls (primitives) and data structures defined by the language runtime support.

Consider compiling libEr code without any integration of the Erbium language within the compiler. Every synchronization or memory operation is represented as a function call. In the best case, the library is compiled simultaneously with the application, providing the compiler with much clearer information of what are in fact these functions. However, as the calls are within different threads and sharing pointers (record data structures), the compiler, similar to any other library calls, is not able to predict any behaviour from the primitive function calls and assume them as containing side effects.

The next section details Erbium as a GCC intermediate representation and how such restrictions can be hidden by exploiting its primitives properties and by adjusting them to analysis and optimizations already available in GCC.

4.3 Erbium in GCC

GCC is one of the most complex compiler frameworks. Modifying GIMPLE middle-end representation to support Erbium primitives would imply many other modifications through affected optimization and analysis passes. Moreover, this approach would make the implementation of Erbium nearly impossible in due time.

Luckily, the Erbium language can be defined as an extension to C, its primitives can easily be expressed with the existing GIMPLE representation. Moreover, as shown in the runtime chapter, the Erbium primitives can be represented as regular function calls, making builtins the best candidates to represent them. Builtins require no modification to GCC's original GIMPLE intermediate representation and passes. The language extensions such as OpenMP [6, 17] and transactional memory [15] support have similar integrations.

The integration of Erbium language in GCC implied extending the front-end phase, simplifying process definition and process instantiation. From several tried alternatives, the best and less obtrusive approach was to define a new function attribute, flagging functions as processes.

```
void process (void *rec) __attribute__ ((process)) { ...  }
```

Any call to a attributed process function is converted to a process instantiation.

For Erbium middle-end and runtime support compatibility reasons, process definitions and all its calls (process instantiations) are transformed during GIMPLE lowering pass. It

```
void __builtin_er_update (void *view, int index);
void __builtin_er_commit (void *view, int index);
void __builtin_er_stall (void *view, int index);
void __builtin_er_release (void *view, int index);
```

Figure 4.7: GCC Erbium synchronization builtins.

```
void * __builtin_er_alloc_record (int elem_size, int buffer_size_in_elems );
void * __builtin_er_alloc_view (int elem_size, int horizon, char view_type);
void __builtin_er_add_registered_views (void *record, int nr_readers, int nr_writers );
void * __builtin_er_get_new_view_id (void *view, char view_type);
void * __builtin_er_connect_registered (void *view, void *record, void *view_id);
void __builtin_er_connect (void *view, void *record);
void __builtin_er_free_view (void *view);
```

Figure 4.8: GCC Erbium record and view initialization and termination builtins.

involves merging all of its parameters into a single data structure collecting the process function parameters into a single memory entity. Moreover, all the process instantiations are converted into calls to specific process instance creation and execution builtins. Figure 4.6 presents such builtins having the same prototype definition as its runtime support presented in Chapter 3.

Figure 4.13 and 4.14 are code examples of the before and after GIMPLE lowering pass.

Synchronization primitives are also implemented through builtin functions (Figure 4.7) similarly to how synchronization primitives are defined in Chapter 3.

Within the compiler representation, both record and view data structures are defined as non typed memory pointers (*void**). In reality, there is no real benefit in defining such data structures precisely, considering its content is only accessed by the runtime support library. No application is assumed to access the data structures directly. Doing so would violate Erbium code portability, considering that view and record data structures, as well as primitives implementations, are defined for particular target architectures or operating system. Moreover, the view and record data structures can have different content for various implementations.

Record and view initialization and termination primitives are presented in Figure 4.8.

Data communication buffer accesses are abstracted using the builtin function: `__builtin_er_occ(void *view, int index)`

This builtin replaces the events access as presented in Chapter 2 using the view with double square brackets (*view*[[*index*]]). This builtin is lowered during compilation into memory accesses (array semantics) where such operations can be further optimized as regular array accesses, in synergy with their surrounding sequential code. Through the compiler execution, the monotonic indexes, used to access the buffer positions, are replaced by modulo operations computing the exact buffer position in memory.

4.3.1 De-obfuscation of Erbium builtins

Although builtins are a easy way to integrating Erbium primitives, builtins are not sufficient by themselves to guarantee the usefulness of the Erbium intermediate representation. Without further integration, builtins are like *extern* function calls and the most pessimistic assumptions are taken, considering the lack of information. The simplistic implementation results in code obfuscation, creating similar problems to code optimizations as any source-to-source compiler generated code would to a traditional mainstream compiler.

In order to allow builtins not to obfuscate code, it is necessary either to lower the builtins at a soon enough compilation stage, or to extend the compiler analysis with precise builtin properties.

Synchronization primitives

Erbium builtins impact many of the existing sequential optimizations. This happens since the compiler makes no assumptions about any prototyped, yet non implemented function (functions implemented outside of the compilation scope, for example runtime library function calls). Non implemented functions are assumed as containing side effects, contaminating or disabling many compiler analysis and optimization transformations.

For example, the compiler cannot predict if this function performs a long jump, has exception handling, or also privately performs some sort of global state change. For this reason, there are very few types of expressions that can be moved across function calls.

In order to avoid code contamination through the Erbium primitives, it is necessary to define such builtins precisely with respect to their internal behaviour.

OpenMP and Trasactional Memory (TM) [15] macros are converted to builtins, as Erbium primitives are. Like so, their builtins also contain side effects. Nevertheless, their lowered builtin functions have too complex semantics, which obfuscate their surrounded code from many of the traditional optimizations, as is shown by Pop et al. [65].

Furthermore, as those builtins are not further detailed through the compiler implementation, they get considered as containing side effects. In any case and considering the semantics of OpenMP and TM primitives, such behaviour might be necessary and intended, to guarantee compilation correctness.

As explained in Chapter 2, Erbium primitives have very simple properties, and contain no dependencies with respect to the sequential code, apart from statements accessing buffers. In other words, Erbium synchronization primitives are only intended to protect buffers content and not any traditional statements, as for example a *mutex* would. Moreover, apart from buffer accesses, every other operations should be movable across a synchronization builtin.

These properties allow Erbium, contrarily to OpenMP and TM, to better define its primitive function "effects", allowing existing optimizations to execute without code obfuscation from the Erbium primitives.

Remove side effects from builtins

In order to define synchronization builtins without side effects, it is necessary to inform the compiler that none of the synchronization builtins has side effects, i.e., the builtin will not contain any strange behaviour, such that switching the order of operations would change the semantics of the application. Such type of side effects can be expected on mutex lock and unlock builtins. Indeed, those primitives are intended to guarantee multiple exclusivity

of their protected code, thus such code cannot move across either lock or unlock builtins. Nevertheless, in case of Erbium builtins, even if its runtime uses mutexes, as long as its correctly implemented, the Erbium builtins can be interpreted as not containing such type of side effects.

The standard way to define side effects free function is through the attribute *pure* or *const*. Nevertheless, Erbium synchronizations cannot be defined as such considering that not all the primitives return values and their arguments content does change during the primitive execution. For example, performing an *update* changes the view sliding window, which can be understood as a change in the view buffer content. Moreover, the only side effect of *update* is to change the content of its views buffer.

Furthermore, both *const* and *pure* attributes expect the function to return a value, its only effect being this returned value. Without a return value, all calls to such functions would be detected as dead-code, considering that no effects were expected to occur to the passed arguments. Thus, *pure* and *const* attributes cannot be used in the context of Erbium primitives.

In order to flag the Erbium builtin calls as side effect free, a new precise attribute type is defined, specifying that the builtin has no global application side effects.

Alias analysis and code motion

Alias analysis has a big influence in code motion decision. Without going in greater implementation detail for code motion passes such as PRE, the rule of thumb is that only non-dependent expressions or statements should be reordered. If a statement uses pointer arithmetics, it is harder to predict its dependencies. Moreover, it is necessary for alias analysis to disambiguate possible aliasing variables, and to guarantee that those pointers cannot possibly be pointing to a dependent reference.

When statements are function calls and such functions expect pointers in their arguments, alias analysis should traverse such function code in order to identify what are the references created by the function. External function definitions do not provide compilers with sufficient information to disambiguate pointer aliasing. Moreover, the worst case scenario is considered and all the pointer arguments are considered referenced as escaped, meaning that those pointers aliasing is impossible to predict (they might alias with anything).

The Erbium primitives and more precisely its synchronization builtins expect a view pointer as their first parameter. As any other builtin, no real definition of the function exists at compilation time.

In GCC, alias analysis is computed with the already referred “set constraints”, which defines the rules for pointer aliasing and allows alias analysis to infer points-to information for every pointer variable. The builtins implementation allows to have precisely defined set constraints such that their pointer arguments are not considered as escaped as traditional functions would be.

Precise implementations for specific builtins are already done for GCC builtins, such as *malloc* and *memcpy*, avoiding the returned memory from *malloc* and both pointers provided to *memcpy* from being classified as escaped. The *malloc* set constraints define that its returned pointer is a newly memory position on heap address space while *memcpy* with both its pointer arguments define that both the passed pointers are neither escaped nor related though this call.

The Erbium primitives have similar implementation, more precisely both record and view

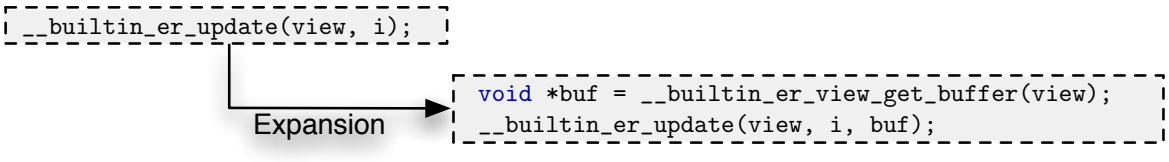


Figure 4.9: Conversion of update primitive, introducing a buffer pointer reference.

allocation should be associated through their returned pointer with a heap allocation and all the other primitives should avoid escaping any of their view and record arguments. Moreover, having specific sets constraints for the Erbium primitives minimizes the compiler pessimism regarding pointer aliasing, by statically defining the pointer aliasing occurring internally in those external functions.

Used or clobbered pointer by builtins

The alias analysis in GCC provides one step further analysis thorough an alias-oracle, serving as an abstraction to perform queries using the points-to information collected by alias analysis. One of these possible queries is to verify if a statement is using or writing (clobbering) a specific memory reference.

Once again, GCC allows builtins to go one step further regarding its builtin primitives specialization. Without such specialization, the builtin is assumed as using or clobbering every pointer variable and only very clear cases where this is not possible are optimized, i.e., the cases where pointer aliasing is impossible.

An implementation example is *memcpy* builtin. GCC can also identify if any pointer is either used and/or clobbered by a specific call. Without such type of analysis, possibly aliasing pointers would be always treated as being used and clobbered by every non defined function. As one should know, *memcpy* transfers n bytes (third argument) from the region pointed by its second argument into the region of the second argument. Consider a pointer v and the following statement $memcpy(d, s)$. If its second argument s is a reference or alias to v, the content pointed by v is used by the call. Similarly, v is clobbered by *memcpy* if d references v. Moreover, in cases where v can potentially be a reference to any of its arguments, it should be considered as such with respect to its optimizations (code motion for example).

In the particular case of Erbium, all the builtins containing view or record pointers should have such specialization. The most interesting case is the *update* and *stall* primitives where buffers should be considered as changing their content with these calls, i.e., the buffer memory is clobbered by these primitive calls.

Nevertheless, in order to verify such clobbering, it is necessary to dereference the view in order to collect the buffer pointer. To facilitate such operation, all the buffer related builtins are converted with an extra pointer argument for the buffer, simplifying its implementation. Moreover, this simplification has no code penalties considering the extra builtin call is defined as *pure* (See Figure 4.9).

Table 4.1 sum-ups all of the alias analysis specializations required for the Erbium builtins relevant for synchronization. For each of the relevant Erbium builtins, the table presents how alias analysis should take into consideration each builtin. More precisely, it gives the set constraints rules (Alias column) and the usage or clobbering for each individual argument.

Builtin	Alias	Uses	Clobbers
void *er_alloc_view(...) void *er_alloc_record(...)	LHS → HEAP	-	LHS
er_connect(*view, *record)	-	record	view
er_connect_registered(*view, *record, *vid)	-	record, vid	view
er_free_view(*view)	-	view	view
er_update(*view, index, *buffer)	-	view	buffer
v = er_occ(*view, index, *buffer)	-	view, buffer	-
er_release(*view, index, *buffer)	-	view	buffer
er_stall(*view, index, *buffer)	-	view, buffer	-
er_occ(*view, index, *buffer) = v	-	view	buffer
er_commit(*view, index, *buffer)	-	view, buffer	-
void *malloc(size)	LHS → HEAP	-	LHS
void *memcpy(*dest, *orig, size)	LHS → dest	orig	dest

Table 4.1: Aliasing, usage and clobbering information for Erbium builtins. The *malloc* and *memcpy* builtins are also included as reference for the existing implementation.

```
void * __builtin_er_view_get_buffer(void *view);
int __builtin_er_view_get_elem_size(void *view);
int __builtin_er_view_get_buf_elem_size(void *view);
```

Figure 4.10: Builtins used in the *occ* primitive conversion.

This set of rules specifies how the Erbium primitives are (un)related to non Erbium code[1], allowing other pointer expressions to be moved across Erbium primitives when not aliasing with the buffer pointers. Moreover, setting the clobbering or uses for each primitive will allow to also reorder Erbium primitives without having to take in consideration its index argument. For example reordering similar primitive call types when not followed by other primitives, such as *occ* operations.

4.3.2 OCC expansion

As previously mentioned, buffer accesses are initially represented by a builtin *__builtin_er_occ* (or occurrence) function, abstracting a view and an monotonic index from the final buffer memory positioning.

Soon enough in the compilation flow, *occ* builtins are lowered into array-like buffer accesses using other builtin functions as getters for the buffer pointer, buffer size, and element size for the particular view. The prototypes for the builtins used in the conversion are presented in Figure 4.10.

Figure 4.11 is the lowering of the higher level abstraction *__builtin_er_occ* function, into the more general buffer memory builtin functions and later array-like buffer access using modulo operation to obtain the buffer memory position for its monotonic index (i).

[1] Although the Erbium synchronization has no direct relation with non Erbium code, it is possible to enforce a dependence through an event access (an *occ*). This is the case if the *occ* is not dead-code which can be enforced by, for example, reading or writing the *occ* from a volatile variable, or using it as part of the process control flow decision.

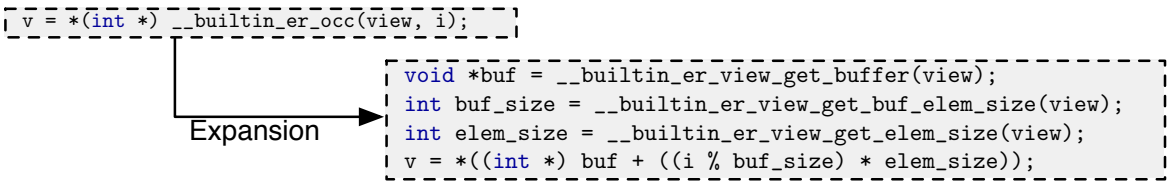


Figure 4.11: Expansion of *occ* primitive into more general primitives and array like buffer accesses.

This lowering creates 3 new variables containing a pointer to the buffer, the buffer capacity in number of elements and the buffer element size. One might consider this approach too expensive considering the increase in number of variables and operations, nevertheless the compiler will do its work by removing redundant calls to those builtins as well as repeated variables.

As previously explained, such builtins degrade performance by disabling optimization such as code motion of itself or any of its surrounding code. However, as a view is only clobbered by the *connect* and $free$ primitives and the view pointer is kept constant between these two calls, the resulted buffer position, size and horizon should only change when executing either of these primitives. Moreover, we can define the three new builtin functions as *const*, providing the compiler with the ability to hoist any of these builtin calls as long as the view pointer value is kept constant. Loop invariant code motion, available through PRE in GCC, is the optimization that detects the redundancy and removes the multiple builtin calls, hoisting such calls immediately after the view to records connection and removing all the redundant calls created through the lowering of the *occ* builtin.

Although the conversion optimizes code, it also introduces an expensive modulo operation in order to predict the buffer position for its monotonic index. Moreover such modulo executes, at least, once per converted monotonic index. To remove such operation, Erbium advocates the use of buffers with powers of two size, where modulo is substitutable by a binary *and* (`&`) with ($buffer\_size - 2$).

$$index \ \% \ size = index \ \& \ (size - 1) \ \textbf{if} \ size = 2^n : n \in \mathbb{N}_0$$

Index overflow, also known as wrap-around indexes and explained in Section 3.2, also enforces buffer sizes to be defined with powers of two.

4.3.3 Call-graph extension - Process Network Graph

The call graph data structure, as previously explained, is a fully directed graph representing functions (nodes) and calls (edges) for the entire compilation unit. GCC supports full usage of such graph data structure through all middle-end optimizations, allowing a quick identification of callees and callers for any function. However, call-graph support is limited to traditional function calls.

Unlike regular function calls, process or thread instantiation involves calling an external runtime threading system function, with unpredictable behaviour for current generation of compilers. Moreover, threading functions are interpreted by compilers as black boxes where

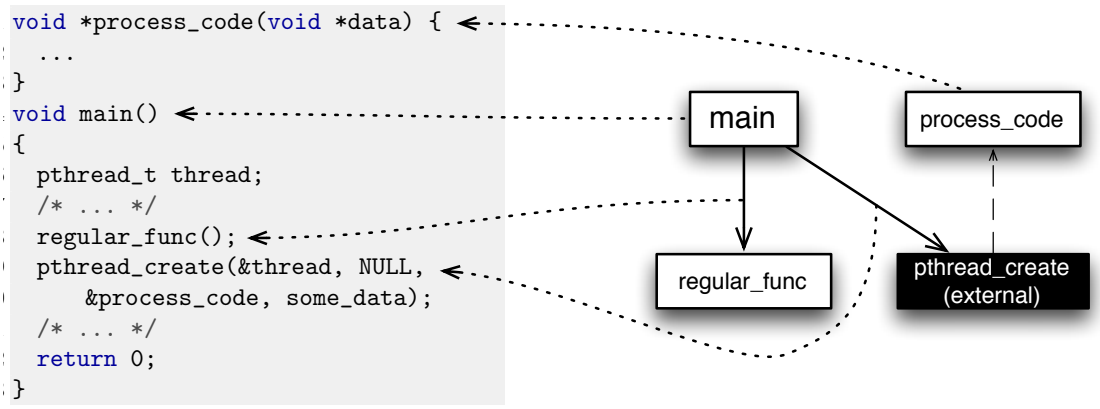


Figure 4.12: Thread creation example using *pthread_create* POSIX function, together with its respective generated callgraph. Solid lines are edges in the call graph. Fine dashed lines are the connection between call graph and code statements. Dashed connection between *pthread_create* and *process_code* node is non existing since *pthread_create* is external to compilation unit and the compiler has no information regarding what it does with its first parameter (*process_code* function pointer).

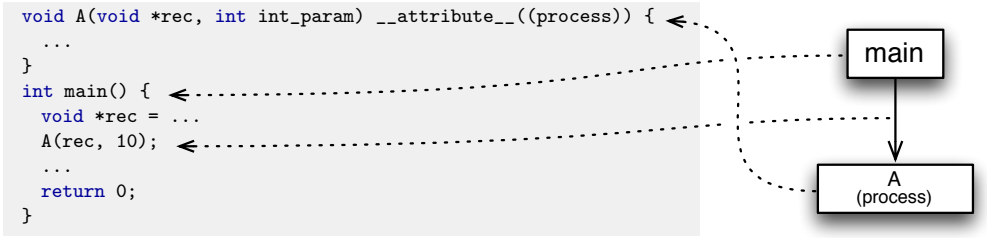


Figure 4.13: Front-end parsable process code example.

no assumptions are made. The lack of information for these external runtime libraries makes it impossible for any compiler to represent threads in its intermediate representation.

Figure 4.12 is a thread creation code using the well-known POSIX runtime library and the resulting call-graph. Considering the lack of information, the instantiated function (*process_code*) is presented has unconnected from the main application code. When compared with the programmer knowledge on *pthread_create* function, the generated graph seems incomplete. For the compiler, functions like *process_code* are alive (and not identified as dead-code), considering the pointer provided as argument in *pthread_create* call.

In Erbium, process instantiation is abstracted by *er_run* and *er_alloc_process_instance* builtins (presented in Figure 4.6), defined in the same way as in Chapter 3.

Using process creation and instantiation builtins, it is possible to define process instances in the existing call graph data structure. The call graph construction can be performed very similarly, and much alike, to what is presented in the inter-procedural data-flow analysis section of [7]

Figures 4.13 and 4.14 are code examples of front-end and middle-end representations and their associated call-graphs, extended with the connection (edge) created during the conversion of call-graph and builtin calls. However, such a call-graph extension, with respect to traditional optimizations, has no usability, since the newly inserted edges cannot be followed

```
struct A_args {
  void *rec;
  int int_param;
};
void A(struct A_args *data) {
  void *rec = data->rec;
  int int_param = data->int_param;
  ...
}

void main() {
  void *rec = ...
  void *A_instance = er_alloc_process_instance(A, sizeof(struct A_args));
  struct A_args *args = *( (struct A_args *) A_instance);
  args->rec = rec;
  args->int_param = 10;
  er_run(A_instance)
  ...
  er_wait_for_processes_end();
  return 0;
}
```

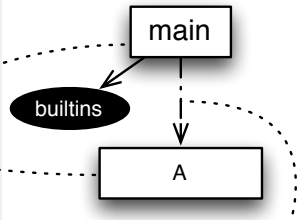


Figure 4.14: Middle-end Erbium converted code from Figure 4.13.

by traditional optimizations. Nevertheless, it is important in the context of new parallel optimizations such as static scheduling, process fusion or any other optimization that requires a clearer understanding of process instantiation.

Such data structure is the foundation of the more expressive Process Network Graph (PNG) concept data structure, advocated and required by the optimizations explained in Chapter 6.

4.4 Process Network Graph concept data structure

Higher level languages provide programmers with simplified abstractions to parallelism problems such as thread creation, process data communication and synchronization. Although they increase programmers productivity, high-level languages abstractions also significantly reduce their expressiveness, especially when compared with the target architecture properties.

Their abstraction level also make them statically predictable. This is the case of the SDF and CSDF computational model languages. On the other hand, a more expressive language such as Erbium, is hardly predictable and very hard to analyze. Each of these expressiveness/abstraction levels is important for its own purposes; the higher-level to analyze and optimize inter-process communications and global application behaviour, the lower-level to tune the process code and synchronizations granularity in a finer way.

However, while transforming the process code, the compiler must be able to validate the transformations through inter-process analysis. This is the main purpose of the Process Network Graph data structure, i.e., to act as an interface between the previous performed higher-level inter-process analysis and the lower-level process optimizations.

The Process Network Graph (PNG) data structure is an extension of the call graph data structure allowing to associate information to both processes and process connection/communication edges. Moreover, apart from collecting process instantiations, the PNG does provide information regarding processes data communication. If two processes communicate, that

connection should be represented as an edge between those two processes nodes and both the GIMPLE nodes representing views and records are bidirectionally bounded to the PNG edges.

The PNG information should be collected during the compiler lowering of high-level languages. Recovering the PNG from Erbium code alone, although not impossible, is mainly limited by the amount of dynamic behaviour exposed by the initial high-level language. Moreover, reversing the PNG information would require expensive analysis, mainly matching the views/processes to communicating records and process instance callers to process instances. In a high-level language, in the worst case, this problem is as difficult as in Erbium, and for many known languages, such as SDF and CSDF language such information is statically provided.

This thesis advocates for the creation of the PNG information at an early compilation stage, if possible during the lowering of the higher-level language source-code into the Erbium IR.

Although, important for the validation of Erbium optimizations, a clear definition of the PNG data structure is beyond the scope of this thesis and although further mentioned within the document, a precise implementation will not be presented. Throughout Chapters 5 and 6, opportunities to collect and use the PNG data structure information are presented and briefly discussed.

The PNG data collection and integration within compiler frameworks is left open for future research.

4.5 Summary

This chapter first presented traditional compilation approaches to parallel languages, typically using higher-level source-to-source transformers adapting their abstracted semantics to lower level sequential languages, using runtime support to implement the original language behaviour, and developed the reasons why such approaches are limitative and obtrusive to mainstream compiler optimizations, and further improvement of parallel code.

A brief detail of the GCC compiler internals was presented together with an extension of GCC, integrating Erbium primitives as de-obfuscated builtins, exploiting GCC's alias analysis through "set constrains" and "use and clobbers" builtin specialization, allowing traditional optimizations to further optimize Erbium codes.

The problems associated with compilers and its ability to the represent higher-evel semantics was presented at the end of the chapter where Process Network Graphs were introduced as a potential data structure unifying both parallel higher-level semantical properties and lower-level Erbium IR expressiveness.

Chapter 5 will now present Streaming OpenMP (a high-level parallel streaming language) and its lowering into Erbium, exposing a few of its streaming properties typically collected in the PNG data structure.

Chapter 6 will use of both the PNG and Erbium builtin primitives to further optimize the application parallelism as well as the low-level synchronization primitives, which were, up to now, impossible to analyze and optimize.

Chapter 5

OMP Compilation to Erbium

High-level languages abstract programmers from many of the complexity of close to hardware parallel programming. The abstractions arise from the hidden lower level semantics involved in the language constructs. Compiler optimizing these languages implies exposing all of the hidden properties (parallelism, synchronization and data communication) in its lowest level form, through a compiler intermediate representation. This is the goal of Erbium's intermediate representation (IR). The code (IR) can then be optimized by exploiting the target architecture properties. Optimizations occur through static analysis, not only using the Erbium intermediate representation but also using the collected information from the initial higher level language abstractions. For example, to adjust the communication granularity based on the available memory local to the process and the synchronization cost, or even fusing tasks, adapting the application parallelism to actual hardware resources.

Extensions of OpenMP have been proposed to support pipeline parallelism and streaming applications [19, 52, 64, 68, 76]. These are promising tradeoffs between declarative abstractions and explicit, target-specific parallelization. However, they suffer from expressiveness limitations. For example, the presented `GNU FMRadio` application in Section 3.4 has been initially parallelized with such approaches, but data parallelism could not easily be expressed. Pop et al. report speedups saturating around 3× on 4-core to 16-core x86-64 platforms [68]. A new proposal for a streaming extension of OpenMP has benefited from our experience with Erbium [66].

This chapter presents Streaming OpenMP (SOMP) language as a high-level language representable in Erbium intermediate representation. Moreover, the high expressiveness of the Erbium IR enables semantical improvements to the SOMP language.

The chapter also details how a language like SOMP is converted to the Erbium IR, briefly explaining the necessary compiler lowering steps, such as conversions between tasks into Erbium processes, and the decoupling of data-communication from synchronization (through the creation of records and views), initially represented (in SOMP) as streams. The conversion section presented in this chapter is the outcome of the development effort done.

5.1 Streaming OpenMP

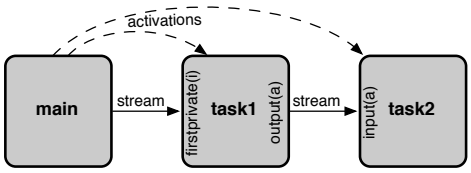
Streaming OpenMP, as the name suggests, is an extension of the OpenMP parallel language. OpenMP [17] is a pragma-based parallel extension of C, C++ and Fortran languages.

Streaming OpenMP extends traditional OpenMP by defining data-flow semantics through

a set of pragma clauses (*input* and *output*) only available when used in a *task* pragma. Using *input* or *output* clause enforces OpenMP tasks to behave in a streaming fashion and communicate through streams.

SOMP, by the time of the developments presented in this thesis, had very limited semantics and properties. Unlike Erbium, its semantics supported only *push* and *pop* operations, executed once per task activation and output or input clause, associated with the particular task pragma.

```
1  main() {
2   int i, a;
3  #pragma parallel private(a, i)
4  #pragma single
5   for(i = 0; i < 100; i++)
6   {
7    {
8  #pragma task firstprivate(i) output(a)
9     {
10     a = f(i);
11    }
12 #pragma task input (a)
13    {
14     g(a);
15    }
16   }
17  }
18 }
```




```
1  stream int s_i, s_a;
2
3  void taskA() {
4    int a;
5    int i = pop(s_i)
6    a = f(i);
7    push(s_a, a);
8  }
9  void taskB() {
10   int a = pop(s_a)
11   g(a);
12 }
13
14 main() {
15  int i, a;
16  for(i = 0; i < 100; i++)
17  {
18 // #pragma task firstprivate(i) output(a)
19     push(s_i, i);
20     libomp_act(taskA);
21 // #pragma task input (a)
22     libomp_act(taskB);
23  }
24 }
```

Figure 5.1: Simple Streaming OpenMP example.

The code in Figure 5.1 is a simplified producer consumer example using SOMP pragmas executing $f.g(i)$ for $i \in 0, 1, ...99$, together with a pseudo translation. Both f and g, when compiled with SOMP support, are pipelined and variables a and i are converted to streams. In the example, one can notice the use of the *parallel* OpenMP pragma followed by a *single* pragma. This approach is just a libgomp work-around, instantiating the multiple threads used for the task executing. It creates multiple threads but also enforces the main thread to execute in just one of the created threads.

Within traditional OpenMP, *#pragma task* defines a task region (function) later executed in parallel. When control-flow reaches this pragma, it requests the task execution (scheduling) within any of the available threads created in *omp parallel*.

Tasks containing *input* or *output* clauses, like in the example, are interpreted as streaming tasks. *Input* and *output* require one or more variable parameters. These variables define which values are communicated as streams. During compilation the variables are substituted by libgomp FIFO buffers, accessed by push and pop operations.

Task activations occur in similar manner as previous standard OpenMP tasks, nevertheless, these tasks are bounded to data availability in the streams used in their *input* clauses. The resulting lowered *pop* operation blocks task execution as long as no data is available at

the respective stream.

The *firstprivate* clause, when used within a streaming task, is lowered to an input streaming buffer, just like the *input* clause. In comparison with *input*, *firstprivate* is not expected to have other streaming task as its stream producer, but instead the data production occurs next to the task pragma, i.e. the pushed stream value is the result of the task parent region and its control-flow. For example, in Figure 5.1, the *firstprivate(i)* clause is produced in the context of the *single* pragma, right where the task pragma is defined. When such pragma is visited, within the outer region control-flow, a new value is pushed into the respective streaming buffer, just before the task is activated (Line 19 in the example conversion in Figure 5.1).

GCC implementation

GCC has one of the oldest implementations of the OpenMP API and is the chosen framework for the Streaming OpenMP conversion.

The Streaming OpenMP conversion is done in two distinct compilation phases, more precisely, the lowering and expansion phases.

The lowering phase is responsible from converting the high-level constructs within the abstract syntax tree into GCC GIMPLE form[1]. It is also responsible for converting pragmas into specific OpenMP related data structures, organizing the pragmas related code into regions, defining the code scopes associated to the pragmas. At the lowering phase, the abstract syntax tree within GCC still has many of the richness of C language including scopes (opening and closing brackets) and pragma information. This compilation phase collects all the necessary information from the pragmas into *omp_region* data structure elements and "lowers" the abstract syntax tree into *GIMPLE* form.

In GIMPLE form, all of the structural information is lost and converted into conditional *gotos*. The *omp_region* data structure stores the hierarchical information available within the lowered OpenMP code. This data structure is created hierarchically based on OpenMP pragmas. As all the OpenMP regions are single entry and single exit code, omp_region data structures keeps track of the referring code by keeping pointers to entry and exit basic blocks, as well as other important basic blocks used for initialization and termination.

The expansion phase takes all the generated regions collected during lowering phase and splits the code regions through different functional units. Within this functional splitting it also generates the glue code necessary for initialization, termination and, in case of Streaming OpenMP, inter-task communication.

The conversion of streaming tasks implies identifying pairs of *input* and *output* clauses, analyzing how tasks communicate and eventually perform the necessary transformations. In case of an $input(a)$ clause, it implies to add code to the beginning of the task and to assign a with the content returned by $pop(stream\_a)$. The clause $output(a)$ implies a similar transformation, but, this time, at the end of the task code and with $push(stream\_a, a)$, pushing the new value of a into the stream. Stream variables are defined and allocated externally to the tasks code (beginning of main function). However, during the expansion phase, these variables are also defined as private for the newly task created function. Please refer to Figure 5.1 as a code example of the initial SOMP conversion.

[1] High-level constructs are any control-flow rich language constructs. Examples of such constructs are the *for* and *while* keywords. The GIMPLE form is limited to conditional gotos.

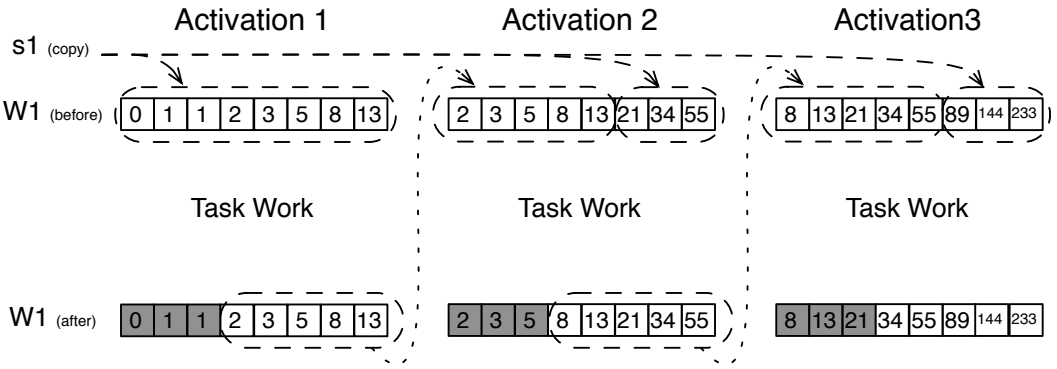


Figure 5.2: Semantics of the new input output clause with respect to window content. The diagram presents three task activations where the sliding window is defined with a size of 8 elements ($int\ W1[8]$) and burst of 3. (clause example: $input(s1 << W1[3])$).

5.2 Streaming OpenMP improvements

Although consistent and safe, Streaming OpenMP is too restrictive and lacks much of Erbium's expressiveness, for example, dynamic production and consumption rates (burst), buffer peek and poke semantics, and the overhead improvements of lightweight persistent tasks.

In an attempt to express features similar to Erbium's, the semantics of *input* and *output* were extended, based on the work of Pop and Cohen [66]. With this extension, tasks are no longer restricted to push and pop properties.

```
#pragma omp task input(s1 >> W1[n]) output(s2 << W2[m])
```

The previous pragma defines sliding-windows *W1* and *W2* for the streams *s1* and *s2*, respectively. The $s1$ and $s2$ streams are defined as regular C variables, but have similar semantics to the earlier presented clause syntax, previously converted into streams. Both $W1$ and $W2$ are defined as arrays of the same type as variables $s1$ and $s2$, respectively. The $>>$ and $<<$ operators are used as syntactic sugar, representing the direction of the copy operation also enforced by the *input* or *output* clauses, allowing to interchange the order of the window and stream operands. For example, `s1 >> W1[n]` is the same as `W1[n] << s1`.

The dynamic rate of consumption or production is defined based on the evolution of the variables n and m, which are provided during task instantiation and remain private to the task in all subsequent task activations. The previous design, through the use of push and pop operations, enforced a static and constant (rate of 1 element) production and consumption rates (burst). This new pragma syntax enables the task code to control its own rate dynamically, based on a task-private burst variable.

Apart from the extensions to the *input* and *output* clauses, code generation was also enhanced to express data parallelism. This is possible considering that Erbium's persistent processes do not depend on explicit activations. Nevertheless, Streaming OpenMP does not include any data distribution / work splitting abstractions and explicit processes must be defined to perform such data split and merges.

Figure 5.2 is a graphical representation for the window $W1$ evolution, from the previous pragma example, where the content of the stream $s1$ is the fibonacci sequence. The window

$W1$ is defined as an array of eight elements, and such array represents the visible elements or the sliding window associated to the respective stream. At the first activation, the content of $W1$, as it is used in an input clause, is loaded with data from the stream $s1$. At this point the task code accesses $W1$ as any traditional array. Once the task code terminates, burst number of elements (n in the pragma example) are discarded from $W1$ and the remaining elements are shifted. At the next task activation the empty windows elements are loaded with new stream ($s1$) data.

The *output* clause has a symmetrical behaviour. The window at first activation contains uninitialized elements which are assigned during task execution. At each activation, burst elements (m) are pushed into the stream and the remaining window array elements are shifted. The next task activation is able to access any data which was not pushed and the last burst elements of the window are once again uninitialized. As the size of the sliding window is constant, the shifted number of elements is easily computable by subtracting the burst for the particular task activation.

Although semantically improved, this version of Streaming OpenMP is not intended as bullet proof, but rather as an example extension, exploiting the Erbium IR expressiveness and presenting a possible language conversion. Moreover, this document is not advocating for such a language improvement on SOMP. SOMP has recently involved into a more mature specification, using an extended/specific Erbium runtime implementation as its conversion target [66].

Through OpenMP lowering and expansion phases all of the static analyzable information from Streaming OpenMP is collected into the previously mentioned Process Network Graph (PNG) data structure.

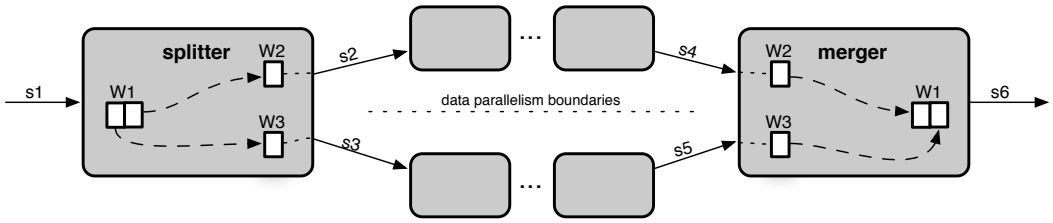
Exploiting data-parallelism

Although the presented extensions to SOMP do not provide any precise abstraction to data parallelism (data splitting), these extensions provide sufficient expressiveness to exploit it.

Figure 5.3 presents both a splitter and merger process, possibly used to distribute data between statically instantiated identical processes or, in other words, data-parallelize the application. The previous SOMP implementation is also able to produce similar data parallelism, however resorting to the less intuitive stream data type manipulations, not allowing any adjusting of the data splitting, i.e., load balance data distribution.

5.3 Conversion into Erbium

The conversion to the Erbium IR allowed Streaming OpenMP further semantical extendability. As these extensions are defined taking in consideration the Erbium language, its constructs closely resemble the Erbium's ones and properties. Moreover, the SOMP streams are converted to records and the windows are converted to views. View horizon is defined based on the window size while the burst value is used to set how many events are released or committed per process iteration (previous task activation).




```
1  int s1, s2, s3;
2  int W1[2], W2[1], W3[1];
3   ...
4  // Splitter
5  #pragma task input(s1 >> W1[2]) \
6               output(s2 << W2[1], s3 << W3[1])
7      {
8          W2[0] = W1[0];
9          W3[0] = W1[1];
10     }
```

```
1  int s4, s5, s6;
2  int W4[1], W5[1], W6[2];
3   ...
4  // Merger
5  #pragma task input(s4 >> W4[1], s5 >> W5[1]) \
6               output(s6 << W6[2])
7      {
8          W6[0] = W4[0];
9          W6[1] = W5[0];
10     }
```

Figure 5.3: SOMP splitter and merger tasks.

5.3.1 SOMP task to Erbium process

Although having streaming semantics, SOMP tasks are not persistently attached into a thread, but instead continuously re-scheduled once an activation is necessary. Erbium processes, on the other have persistent semantics, and are defined as persistent, only terminating once no more work is required, i.e., no more data is available for consumption or can be produced. Considering its streaming semantics, SOMP tasks are convertible into Erbium processes, as termination can be detected through streams.

In the presented version, and as a simplification, tasks are defined in a static manner and only a single process is defined per task pragma declaration, independently of the surrounding code. Although currently defined as static, there is no real constraint from Erbium to express more dynamic semantics. This version is easily extensible to a less restrictive abstraction, allowing parent pragmas to impact the meaning of the task pragma. For example, a task pragma when inside a pragma parallel region could imply that more then one instance of the same task would be instantiated. Nevertheless, to make use of such multiple instances the language would also have to address data distribution, i.e., to explicitly define how data is partitioned through the now fully parallel tasks.

In the context of this document such data parallelism would increase the translation complexity above the chapter intentions and for that reason the semantics of SOMP has been simplified.

Like previous mentioned in Chapter 4, high-level languages contain high-level semantical constructs, simplifying code behaviour prediction and static compiler analysis. In SOMP, the pragmas allow the compiler to collect rich information behavioral information to construct the PNG data structure. Examples of such information is the existence of a full application graph containing all the connected process instances. Moreover, its constant horizon sizes are easily recovered and eventually used to compute large enough buffer sizes, as well as collect burst sizes for each of the independent SOMP tasks, in case bursts are constant and considering the latest syntax of the SOMP clauses.

Figure 5.4 is an conversion example of a simple producer consumer application. Like mentioned, any previous stream variable is converted into a record and processes get instantiated once per streaming task pragma in the code. The presented conversion makes use of

```
1  main() {
2    int i, a;
3    for(i = 0; i < 100; i++)
4    {
5  #pragma task firstprivate(i) output(a)
6      {
7        a = f(i);
8      }
9  #pragma task input(a)
10     {
11       g(a);
12     }
13   }
14   return 0;
15 }
```

```
1  main() {
2    int i, a;
3    record int r_i;
4    record int r_a;
5
6    void *png_context = png_new_context();
7
8    add registered views(r_i, png_nr_readers(r_i), png_nr_writers(r_i));
9    add registered views(r_a, png_nr_readers(r_a), png_nr_writers(r_a));
10
11   for(i = 0; i < 100; i++)
12   {
13     // firstprivate (i) expansion code
14     if (!png_running(png_context, &A))
15       RUN A(r_i, r_a);
16
17     if (!png_running(png_context, &B))
18       RUN B(r_a);
19   }
20    wait_for_instance_list_end ();
21 }
```

Figure 5.4: Conversion of a SOMP main function; original code (left) and Erbium version (right).

PNG related prototype functions, abstracting the code from the PNG collected information. These functions are not part of the converted code but rather a simplification of the necessary variables and condition checks required by the conversion.

These functions are:

- *png_nr_readers* and *png_nr_writers* is a counter for the number of inputs and outputs clauses used for a specific variable, now represented as a record. The number of writers should always be 1 for this particular implementation as multiple writers for the same stream are no supported.
- *png_new_context* serves as a marker for a OpenMP region entry.
- *png_running* returns true if a particular process is already instantiated for the particular context (OpenMP region). In this version, process instantiation is protected using a single boolean variable, initialized to false, next to the *png_new_context* builtin. Once the process is instantiated, such a variable is set to true, avoiding further process instantiations.

As the process instantiation condition is only true for the first loop iteration, the generated condition is most likely considered as loop invariant and the compiler, through its normal work-flow, hoists the process instantiation out of the loop. Nevertheless, considering its statically-predictable process instantiation, one can deduce that code generation directly performs the process instantiation at the beginning of the main function.

5.3.2 Task to process

The translation of the main transforms all streams into records and performs any of the necessary process instantiations. The expansion phase traverses all the input and output clauses, creating a view for each of the clauses. The views are initialized and connected to the respective communicating records. Please refer to the previous example, where the record is already initialized and expects the precise amount of registered reader and writer view connections.

Let us consider the particular case of an input clause. As mentioned before, a window in SOMP semantics is similar to an array with fix amount of elements, and at each task

activation the array is loaded with stream data. In the Erbium case, this behaviour is replicated with the *update* primitive. Updating by the same index as the size of the window, the same amount of events (data elements) become accessible in the view sliding window, i.e., the number of events accessible is defined based on the view horizon size, which must be equal or higher to the window array size.

After task code execution, n number of elements are discarded from the window array, where n is the burst size. A process executes a *release* with *burst* number of events to do the same.

For the *output* clause, the beginning of a task activation involves executing the *stall* primitive, guaranteeing that sufficiently many (equal to the window size) events are accessible for writing. At the end of each task activation, the process *commits*, "writing" the committed events as available for consumption.

As primitives work with monotonic indexes, proceeding next iterations have to increment the primitives call indexes by the burst size. The burst variable can directly be manipulated by the task code, allowing the task to individually modify the production and consumption burst sizes. The *burst* used variables are converted as traditional OpenMP *private* clause variables.

Figure 5.5 an example of the translation of a task into an Erbium process where the explained translation steps are presented. The task code is omitted in the example and only the relevant translation parts are presented.

```
1  int i, a;
2  int W1[W1_SIZE];
3  int W2[W2_SIZE]
4  // ...
5  #pragma task input(i >> W1[n]) \
6                output(a << W2[m])
7      {
8        // ... task code ...
9      }
```

```
1  PROCESS task1(record int r_i, int n, record int r_a, int m)
2  {
3    // Define views and index induction variables for each input and
4    // output clause in task pragma
5    view int v_i;
6    view int v_a;
7    int i_i = 0;
8    int i_a = 0;
9    bool terminate = false;
10
11   // Initialize all views
12   INIT_VIEW(v_i, READER, W1_SIZE);
13   INIT_VIEW(v_a, WRITER, W2_SIZE);
14
15   // Connect views to records in arguments
16   CONNECT_REGISTERED(v_i, r_i);
17   CONNECT_REGISTERED(v_a, r_a);
18
19   // Loop
20   while(true) {
21     // Update on all reader views
22     terminate |= (UPDATE(v_i, i_i + W1_SIZE) != (i_i + W1_SIZE));
23
24     // Break if any of the updates did not return expected index
25     if(terminate == true)
26       break;
27
28     // Stall all writer views
29     STALL(v_a, i_a + W2_SIZE);
30
31     // Do task work by substituting original variable with view
32     // accesses.
33     // ... task code ...
34
35     // Release and commit views
36     RELEASE(v_i, i_i + n);
37     COMMIT(v_a, i_a + m);
38
39     // Increment induction variables
40     i_i += n;
41     i_a += m;
42   }
43
44   FREE_VIEW(v_i);
45   FREE_VIEW(v_a);
46 }
```

Figure 5.5: Example conversion of a producer consumer SOMP task (left) to Erbium IR (right).

5.3.3 Termination

SOMP task termination occurs through the absence of activations from its parent thread acting as a scheduler. As SOMP tasks are converted to persistent processes, such behaviour is not implicitly available or possible. One could create a record per task that could act as the task activator, allowing the main thread to similarly request task activations (by committing to the record). However, such approach would enforce too much overhead, considering the extra record and synchronization between the main thread and the record process.

Erbium processes are activated through data availability within its reader views. Moreover, we can assume that the *firstprivate* clause represents the task activator. By providing an external to the processes stream writer, such as the loop counter i in the example of Figure 5.4, the exact same behaviour as previous SOMP implementation is obtained, i.e., the process iterates as often as the previous tasks, keeping the exact same behaviour/state as the initial task activation scheme. The consequent (directly connected) processes (any process without the $firstprivate$ clause) are activated, not through an implicit task activation but, as soon as the process notices new data is available through its reader views (*update* primitive). A task activation is a single execution of the OpenMP task pragma region, and happens when the application control-flow reaches the entrance of the specific task pragma region.

Every SOMP converted process, as in Figure 5.5, contains a boolean variable initialized to false. This variable allows the detection of an input clause data termination and consequent process termination.

The task termination detection is based on the termination detection of any of the process reader views. As soon as any of the process views have returned a non consistent value, i.e., the result of update is different from its index parameter (Line 24), the process *break*s from the infinite loop (line 27), freeing all its views and returning from the process code.

5.3.4 Task code adaptation

SOMP tasks directly operate the window or stream variables as an abstraction to parallel data streamization. In previous presented conversion neither the stream or window variables values are initialized at each process iteration. Initializing those values implies the extra overhead of setting them, copying from the view buffer at each iteration. In the case of a stream variable, as it is is defined as a simple variable access, such redundancy is easily detectable by the compiler and eventually removed through copy propagation. Nevertheless, in the case of windows, as windows are defined as arrays, this optimization is unlikely to occur, considering the possible dynamic behaviour of its accesses (dynamic index values).

Minimising the data access cost, implies that during the expansion phase we traverse through all the task statements, substituting any window and stream variable references by a direct access to the view events data (using an *occ* primitive call). As *occ* (defined as a builtin function) returns the memory address of a particular event data, a new temporary pointer variable is inserted before any original code access to a window or stream.

Once all the windows and stream accesses are substituted by *occ* calls, the expansion phase into Erbium processes terminates.

Figure 5.6 is a stream and window variable conversion example, more precisely, the conversion of b and $W1[j]$ to *occ* primitive calls.

```
#pragma task input(a >> W1[1]) \
             output(b) \
             private(j)
    {
      float t = 0;
      for(j = 0 ; j < size; j++)
        t += W1[j];
      b = t / size;
    }
```

```
process task1(record int r_a, record int r_b, int size)
{
  view float v_a, v_b; // Views
  int vi_a, vi_b;      // Index variable
  // Init views and connect
  while(1) {
    // Stall and Update

      float t = 0;
      for(j = 0 ; j < size; j++)
      {
        float *tmp1 = occ(v_a, vi_a +1 +j);
        t += *tmp1;
      }
      float *tmp2 = occ(v_b, vi_b +1 +0);
      *tmp2 = t / size;

    // Commit and Release
    va_i += 1; va_b += 1;
  }
  // Free Views
}
```

Figure 5.6: Task code substitution of all window and stream variables by *occ* Erbium's primitive calls.

5.4 Summary

We presented Streaming OpenMP as an illustration of the conversion of a data-flow streaming language to the Erbium intermediate representation. Its high-level semantics provides compilers with rich properties with respect to the applications behaviour and their data flow. Moreover and considering Erbium's expressiveness, Streaming OpenMP was enhanced with dynamic communication bursts and the multiple producers and consumers communication.

The conversion process was explained through the partitioning of its different components, such as the task pragma translation into process instantiation, SOMP short life tasks into Erbium persistent processes and the internal conversion of SOMP stream and windows into direct view events accesses (*occ* builtins).

Chapter 6 is now the continuation of Chapter 4 (where Erbium is integrated into the GCC internals), presenting several optimizations to the Erbium IR. The collected PNG information, extracted during the high-level language lowering, is exploited in the next chapter to validate code transformations and allowing further code optimizations.

Chapter 6

Optimizations

The Erbium intermediate representation, as explained in Chapter 4, brings no support for parallel optimizations. Instead, it presents a non obfuscating integration of the Erbium primitives in GCC, allowing traditional optimizations to optimize not related to or dependent on Erbium. Nothing was presented with respect to the primitives optimizations.

Also in Chapter 4, the Process Network Graph (PNG) was presented as a complement to the low-level Erbium semantics, where the PNG collects additional information available from the static analysis of higher-level languages. For this reason, many of the Erbium-level transformations presented in this chapter are dependent on the PNG data structure.

High-level languages vs. Erbium

Figure 6.1 is an application graph example, containing processes, views and record entities and its connectivity. This graph can be extracted from Erbium code analysis, but as expressive as the Erbium language is, with its dynamic properties, it can occur that such analysis is impossible, for example, in scenarios where any of its entities (process, views and records) cannot be statically related to its initialization statement. Such complex applications can be originated from higher level languages in which such dynamic behaviour is clearly defined and can be statically predicted without relying on complex code analysis. This information must be recovered earlier during the language lowering phase and provided to the compiler, either through code annotations, in case of source-to-source code transformation, or directly collected in the front-end code conversion. The PNG is the compiler data structure that collects this high-level language information.

Static Data-Flow (SDF) languages are an example of such information loss through Erbium translation. Once converted, information is hidden within all the Erbium semantical expressiveness. The PNG provides an extra manner to collect the original language constructors information, providing hints of the initial intended application behaviour without requiring a full reverse engineering of the Erbium primitives. As presented later, many inter-process optimizations are currently only possible if the higher level languages provide enough data through the PNG.

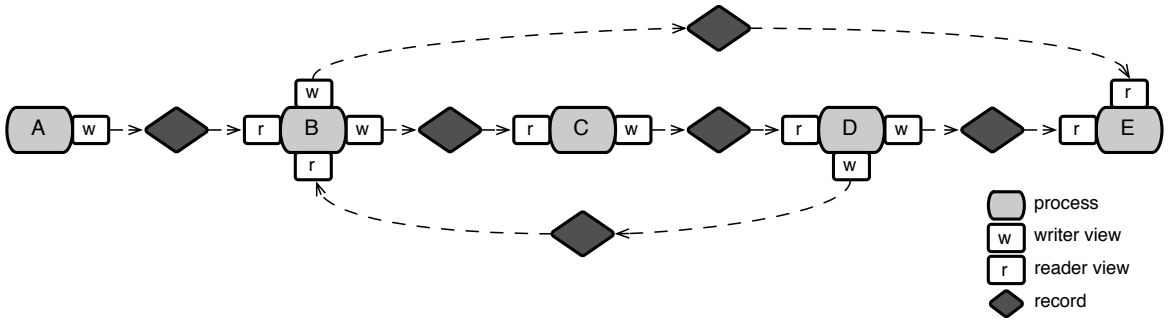


Figure 6.1: Erbium's application diagram.

6.1 Erbium language component dependencies

Compilation implies a progressively lowering of the application language into the target architecture assembly code. The lowering occurs in distinct phases, each further reducing original abstractions available in the initial language. Through its progress, the compiler simultaneously gains and loses its ability to perform certain analysis and optimizations by the succeeding lower-level representations, hiding the initial language semantics in the low-level intermediate ones.

With the introduction of Erbium as an intermediate representation, compilers are yet exposed to several other layers of abstraction, this time related to the required Erbium lowering, more precisely, the conversions presented in Chapter 4 referring to processes and run keywords (attributes) and *occ* operations.

Erbium IR can be classified with three distinctive abstraction levels:

- Highest level (coarsest grain) refers to inter process synchronization and communication in the perspective of a full application. This level of abstraction contains information of the specific records that produce a connection, buffer locations, sizes, as well as the connection graph.

- A finer level including all the builtin synchronizations, the view connection and disconnection calls, embedded in the general control-flow of a process code. Such level includes all the intra-process optimizations, such as code motion or the elimination of spurious synchronization calls, for example, when multiple same type synchronization primitives are executed without a counter part primitive (multiple updates without a release in between). Many times, in order to validate possible inter-process optimizations, it is necessary to access boundary level information (the PNG data structure) verifying optimizations local to the process against the global application semantics.

- The finest level refers to process code optimizations, such as promotion to registers or the vectorization of view events accesses (*occ* operations) or non Erbium dependent code.

Figure 6.2 presents a diagram presenting the three abstraction levels. The left size diagram is the representation of the higher abstraction level, containing information of the application different process instances and their relations. Each diagram component is linked to the precise builtin call related to the instance. The information available at this abstraction level

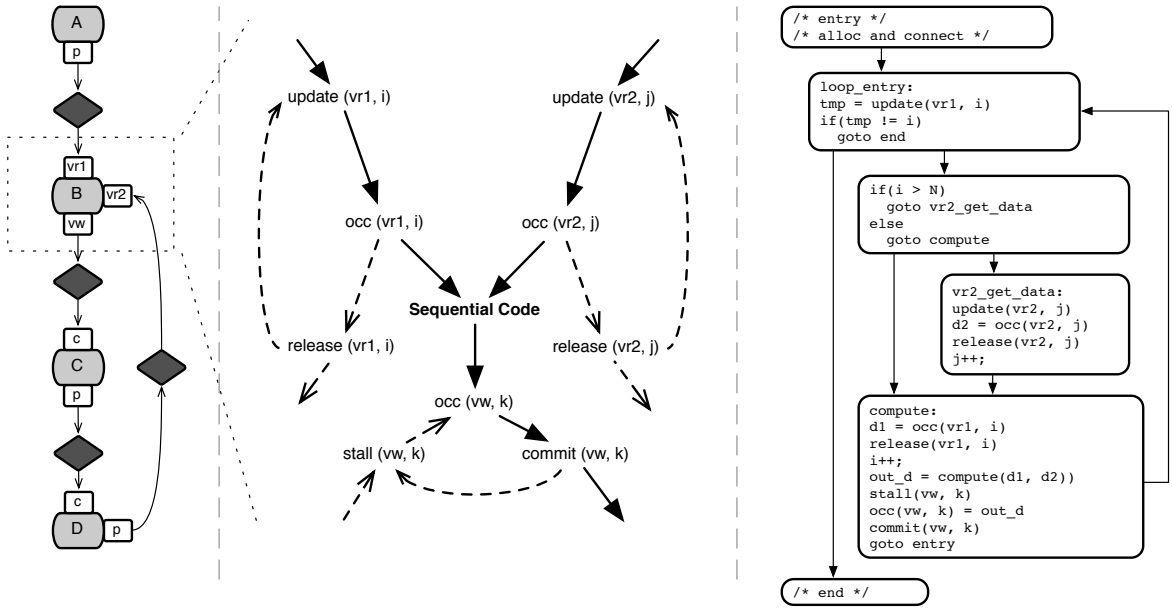


Figure 6.2: Three levels of abstraction within Erbium applications. The two right side diagrams refer to the process B (left diagram) in the more detailed abstraction levels.

defines what the PNG data structure represents. Properties such as the rate of communication and buffer sizes for the channels, although also available in the PNG, are not presented in the figure.

Data communication and synchronization information is not easily obtainable through reverse engineering of the Erbium code. As higher level languages are less expressive than Erbium, the PNG information can more easily be collected during the language conversion. The SDF computational model languages are one example of the languages where the rate of communication between the processes is statically provided, allowing to much easily deduce application behaviour and, for that matter, allows compilers to validate and optimize inter-process communications.

The second level of abstraction (middle diagram in Figure 6.2) contains the synchronization details for each precise process. At this abstraction level, compilers perform analysis over Erbium synchronizations and *occ* builtins still using monotonic indexes, allowing for example to identify builtin calls motion opportunities and redundancies. In order to clearly analyze the synchronization builtin dependencies, this level should rely on previous level information guaranteeing the correctness for its own code transformations.

The last abstraction level is the code composing the process function. At this compilation stage, *occ* builtins have been converted into memory operations, losing its monotonic index into buffer memory positions, using the modulo operations with the buffer sizes, as explained in Chapter 4. In any case, thanks to the changes to alias analysis presented in Chapter 4, the remaining Erbium builtins do not obfuscate traditional compiler optimizations and allow non data-dependent operations to move across builtins, apart from the redundancy detection of buffer accesses previously converted from the *occ* primitive calls.

Alias analysis, by itself, does not provide sufficient meaning to the Erbium builtins for the compilers to be able to optimize synchronizations and application sequential code. For compilers to perform code transformations, the language dependencies and the set of trans-

formation heuristics must be precisely defined such that the compilers can understand the relation between the Erbium primitives and the impact of any of its code transformations. Erbium code dependencies are also studied in the three distinct levels, more precisely the inter-process, synchronization/data communication and code optimization level.

The inter-process dependencies are mainly relevant for application wide optimizations such as process fusion, record merging and static scheduling. The synchronization level is used for process specific optimizations such as synchronization and data-communication code motion, spurious/redundant synchronization elimination and blocking — although still using inter-process dependencies to validate the correctness of the possible optimizations, i.e., taking into consideration global integration of the process before applying any transformation. The last level refers to the traditional non Erbium code or, in other words, the existing compiler optimizations. The Erbium data communication (buffer accesses) benefits from this level through traditional partial redundancy elimination, dead-code elimination, or even register promotion of intermediate uses of buffers. Vectorization can also significantly improve the sequential process code.

6.1.1 Inter-process dependencies

Erbium processes by itself are independent from each other. Process instantiation does not imply any dependency. Inter-process dependencies are defined through the usage of record and view data structures.

Erbium abstractions do not provide the necessary predictability to collect the required information and statically deduce inter-process dependencies. Although possible, inter-process dependencies analysis are out of the scope of this thesis studies. Moreover and in case such dependencies are available by the higher level language, this thesis advocates the use of such information as a way to collect and to create the PNG data structure.

Process instantiation

The Erbium language supports dynamic process instantiation. Processes are instantiated through the Erbium builtin *run*, allowing to embed process instantiations in complicated control-flow, or even inside other Erbium processes. This dynamic behaviour makes it very hard or even impossible to predict application behaviour. Nevertheless, we believe such analysis can be performed as long as not very "deep" dynamic semantics are used, such as random decision.

A process instantiation graph can be constructed, much like the call-graph is created, as mentioned in Chapter 4. In any case, context sensitive information, such as the multiple instantiations created by loop iterations on a single *run* call, is not represented in such a graph. To collect context sensitive information, the compiler should rely on the high-level language semantics to recover the contextual information.

A process (P) defines an algorithm bounded by the Erbium rules, communicating through records which are provided as arguments in the process instantiation. A single process can be instantiated multiple times.

Each *run* $P(args)$ call executed in the application control-flow creates a new process instance (I_n^P) of the process (P), being n a sequential number identifying all of the process P instances.

$$I_n^P = (P, \{arg_0^n, arg_1^n, \cdots, arg_m^n\}) \tag{6.1}$$

Also an instance I_n^P owns a set of inherited views $v_0^{I_n}, \cdots, v_q^{I_n}$, as defined in the process code (P).

Process instance dependencies

It is also important to understand the relation between different processes and their instances, for example, to understand the relation between processes, both at the highest abstraction level (analysing the connection between records and views) and at the finer abstraction level (analysing the channel (buffer) sizes and rates of communication).

Each connection or instance, through analysis, is associated with a set of properties supporting decision for inter-process optimizations, or even validating intra-process optimizations against invalid or deadlocking transformations.

If a process instance $I_1^{P_1}$ owns a writer view $(v_0^{I_1})$ and in P_1 code, v_0 connects as a writer view to one of its arguments, the record r (also identified as $arg_0^{I_1}$), one can say that $v_0^{I_1}$ connects to r $(v_0^{I_1} \rightarrow r)$. Similarly, other process instance with reader view $(v_0^{I_2})$ connects to the same record $(r \rightarrow v_0^{I_2})$. The view is placed on the right hand side of the arrow distinguishing between reader or writer connections.

A process instance is connected to all its defining (owning) views based on its primitive code usage and view type. Moreover and considering the association, both instances and processes can be transitively connected, i.e., connected through the record. For example, two process instances (I_a, I_b) are said **connected** if each contains a view connected to the same record, such that one is a writer and the other is a reader, or in other words, both instances own distinct transitively connected views.

$$I_a^{P_1} \xrightarrow{+} I_b^{P_2} \; if \; \exists m, n \in \mathbb{N}_0 : v_m^{I_a} \xrightarrow{+} v_n^{I_b}$$

Moreover, any two processes have a **connecting path** if there is a chance there are processes instances that connect to the same records, i.e., there is at least one instance of each process with transitively connected views.

$$P_1 \xrightarrow{+} P_2 \; if \; \exists a, b \in \mathbb{N}_0 : I_a^{P_1} \xrightarrow{+} I_b^{P_2}$$

Understanding the inter-process relationship is important to the inter-process compiler optimizations. Process fusion is such an optimization as explained later in the chapter. Moreover, such relations are similarly important for intra-process optimizations. For example, to perform process blocking (increase process burst by aggregation of multiple process iterations), it is necessary to identify connecting processes, as well as other connecting properties such as its buffer sizes and rates of communication.

In a multiple producer and consumer record application, the multiple producers or consumers are also dependent on each other. This type of dependency does not imply a data communication but rather the execution pace of each of the dependent processes. If any of these process instances stop consuming or producing, the dependent processes can starve considering its existing progress dependency. Such type of dependency is important to consider for example load balancing and statically scheduling optimizations.

Relevant inter-process communication properties

Apart from process (task) graph connectivity and instantiation relations, every connected process has less noticeable yet very important properties.

The **buffer size** is one such property. Each record is associated with a record buffer and for the Erbium IR perspective it is considered as shared by all the views. That is not always the case, depending on the target architecture and runtime implementation, nevertheless in the compiler contexts, the final result is negligible considering the runtime is the entity implementing such abstraction.

The buffer size prediction is important to validate process optimizations. Moreover, buffer size adjustment/computation is relevant for the final performance of the application, with respect to memory sizes, memory latency, cache and cache-line sizes (local private memories in case of distributed memories).

The **burst** (or rate) is the number of indexes updated and or committed without the existence of the *stall* and *release* back-pressure primitives. In other words, it is the number of newly indexes available through process iteration if we consider a single synchronization primitive call per process loop. The burst was previously explained in the context of Streaming OpenMP language in Chapter 5.

The **slack** is the maximum index distance between the same process *update* and *commit* primitives, such that, independently of the process execution the *update* primitive does not deadlock. The index difference between *commit* and *update* is only relevant if there is exist an dependence between the *commit* view and the *update* one, i.e. there is an extenal to the process dependency, i.e. a self dependency. A process instance is self dependent if it exists a transitive connection path from any of its process writer views to any of its reader views, i.e., the process is in a loop when analysing the connection graph. Slack value is not unique through all the cycle path processes but instead specifically computed for the specific process and all its pairs of reader and writer views in the loop. *Stall* and *release* primitives have the same type of inter-process dependencies and must also respect the slack distance.

Any self dependency process instance path contains at least one process instance that introduces a consumption delay. A process contains a consumption delay if it commits any data before it tries to consume. The delay defines an initial slack value. The delay distance is not a language property but rather an application algorithm requirement. For instance, h264 decoder implies a self dependence by decoding each macro-block based on the neighbour previously decoded macro-block. In this case, there is a consumption delay between the decoding writer view and the previous decoded frames consumer views. It is undesirable that one of the processes involved in the decoding tries to update for more then one macro-block, since it would result in a deadlock.

Another property contributing to the slack is the burst of all the reader and writer views involved in the cycle path. Depending on the burst rates of each independent process in the cycle path, the slack value can simply be increased or decreased, or in more complicated cases even become of polynomial form, relative to the primitives index. This is the case when the consumption and production rates differ between process iterations, as a result of code control-flow.

In the context of this thesis, we assume that the slack is computable and always constant. Slack computation relates to deadlock detection algorithms widely studied in the context of SDF models of computation, simplified by its constant burst defined for each task. Slack computation should be a job for such an algorithm, executed during the Erbium's higher

```
1  PROCESS A(...) {
2    ...
3    while(true)
4    {
5      STALL(vw1, i)
6      if(i > 5)
7      {
8        UPDATE(vr1, i - 5);
9        // do_something
10       RELEASE(vr1, i - 5);
11     }
12     else
13     {
14       // do_something
15     }
16     COMMIT(vw1, i);
17     i++;
18   }
19   ...
20 }
```

```
1  PROCESS B(...) {
2    int i = 0;
3    ...
4    while(true)
5    {
6      UPDATE(vr2, i + 2)
7      STALL(vw2, i + 1)
8
9        // do_something
10
11     COMMIT(vw2, i + 1)
12     RELEASE(vw2, i + 1)
13     i++;
14   }
15   ...
16 }
```

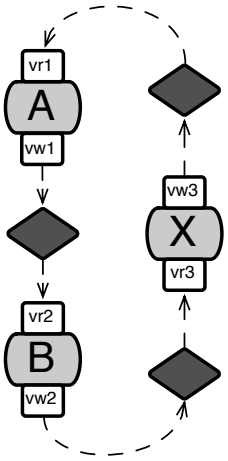


Figure 6.3: Small code example of a possibly self dependent process.

abstraction layer (high-level language analysis) and exposed to lower-level Erbium IR through the PNG data structure.

Later in the chapter, we will present how the slack is used to detect invalid process transformations and avoid deadlocks.

Figure 6.3 is a small code example where three processes are connected in a communication cycle. Non-cycle connections are not present in the example diagram. Both A and B are presented with code.

A programmer is requested to optimize by hand the code of the unknown process (X). As she increased the burst for both the writer and reader views, reducing the amount of synchronization, she realized the application started to deadlock. This is the case because she increased consumption burst above the allowed slack for the cycle transitive connection ($vw3 \xrightarrow{+} vr3$). As the process is integrated in a communication cycle, the programmer must analyze the code of the other processes involved in the cycle, identifying the slack for its process (minimum allowed distance between its update and commit primitives).

To compute the slack for a specific transitive connection path, it is necessary to traverse all the processes in the path, analyzing on a per process iteration the existing differences between consumption and production, i.e., the index differences between update and commit primitives.

By traversing all the processes in the path $vw3 \xrightarrow{+} vr3$, we first visit the process A. This process introduces a delay between production and consumption. By analyzing the code, it is possible to identify that update and commit are always separated by a distance of 5. In the iteration where the process commits the event 6 the process has updated the index 1. Moreover the slack imposed by process A is 5 $(6 - 1)$.

In the process B, on the other hand, for each iteration, it updates by $i + 2$ while commit does it for $i + 1$, moreover process B reduces the slack by 1 event $(i + 1 - (i + 2) = -1)$.

The final slack for $vw3 \xrightarrow{+} vr3$ is of 4 $(5 + (-1))$.

The programmer now knows how to protect his application from deadlocks. The process (X) should never update an index bigger than 4 events of its last commit, otherwise this process instance deadlocks at update call.

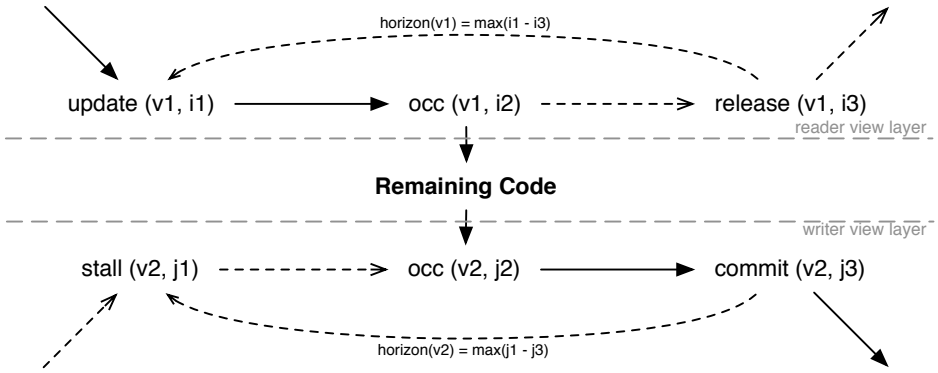


Figure 6.4: Builtin dependencies diagram.

The above properties are part of deeper high-level language analysis and neither reversing or optimizing its value is part of this thesis contributions, or was further studied.

6.1.2 Data communication and synchronization dependencies internal to process

As previous explained Erbium primitives have more refined dependencies then builtins by themselves can represent. Unlike other compiler builtins, Erbium builtins are implemented as side effects free with respect to all non Erbium specific code, as presented in Chapter 4.

Erbium synchronization builtins and view data accesses depend on each other only if their index arguments are associated. More precisely, not every view data access is dependent on all Erbium synchronization primitives and vice versa, but on their respective view calls and only if their monotonic indexes are within the same ranges (inside the view sliding window), respecting the view horizon.

In order to guarantee deterministic results from Erbium *occ* operations, it is necessary for the respective dependent synchronizations to occur before (in case of *stall* and *update*) and after (for *commit* and *release*) the *occ* primitives.

Erbium primitives definitions have no direct dependency to non Erbium code. Primitives only impact the view sliding window, i.e. the view visibility of the record buffers data. The relation between Erbium and the remaining code always occurs in the form of a *occ* operation writing or reading from the accessible view sliding window events. Such access to the view buffers and the buffer dependence to the primitives creates a chain of dependencies between the non Erbium code, buffers and its synchronization primitives.

Figure 6.4 presents a diagram for such dependencies. Solid lines represent all the dependencies associated with data communication through Erbium buffers. Dashed lines are the remaining existing dependencies between the different primitives, not necessarily protecting any data communication but instead guaranteeing correctness of the record resources availability. In any case, not respecting this dependencies breaks application determinism or even produces deadlocks depending on the runtime implementation.

As previously mentioned, the *occ* operation enforces a data dependence between non Erbium code and Erbium data communication and synchronization primitives, either by reading or writing to the view connected record buffer.

The figure graphically demonstrates the actual dependencies from Erbium primitives, as

well as how the Erbium synchronization code relates to the remaining of the code through data dependencies. Nevertheless, not always these dependencies exist for any two primitives, for example, not always an *occ* is dependent on its last predecessor *update*, but only if such primitives relate through their index argument.

Through this document, Erbium dependencies are presented using the delta (δ) character, as commonly used in data dependency equations in the literature. Erbium synchronizations do not represent any variable value dependency, but rather more subtle properties based on the actual operation and its index argument.

Any two Erbium operations are dependent ($a\ \delta\ b$) if and only if a precedes b in control-flow ($a \prec b$) and the operation performed by a is necessary before b to preserve the semantical meaning of b. Such dependencies rules do not in any way guarantee that the application is deterministic, but instead that the application semantics does not change. If the application is non deterministic from the start, it is maintained as initially designed.

The Erbium primitives semantics are directly associated with the primitives index argument value. For example, an *update* primitive call is only needed (or dependent) for any *occ* (view event data access) if and only if the *occ* index argument is within the range of the events visible in the view sliding window expanded by the *update*, or in other words:

$$update(v, i_u)\ \delta\ occ(v, i_o)\ \textbf{if}\ 0 \leq i_u - i_o < horizon(v)$$

This equation represents the only possible relation between any similar view *update* and *occ* operation.

As *update* is a dependency source of *occ* based on its index argument, *release* is dependency sink for *occ* operations. Every *occ* is a dependency source to *release* if the release index argument is at least the *occ* index, or in other words:

$$occ(v, i_o)\ \delta\ release(v, i_r)\ \textbf{if}\ i_o \leq i_r$$

Stall and *commit* have symmetrical dependencies to *update* and *release* with respect to *occ*s.

$$\begin{aligned} stall(v, i_s)\ \delta\ occ(v, i_o)\ \textbf{if} \quad & 0 \leq i_s - i_o < horizon(v) \\ occ(v, i_o)\ \delta\ commit(v, i_c)\ \textbf{if} \quad & i_o \leq i_c \end{aligned}$$

Occ-to-Occ dependencies

Occ operations have no direct relation or dependency between one another. More precisely reader view *occ*s are totally unrelated to one another as well as well as writer view *occ*s. Nevertheless two *occ* operations are dependent if the process code enforces it through data dependencies.

$$occ(v_r, i_1)\ \delta\ occ(v_w, i_2) \quad \textbf{if} \quad a \leftarrow occ(v_r, i_1) \wedge occ(v_w, i_2) \leftarrow b \wedge a\ \delta\ b$$

Preserving view horizon size

Although the presented equations represent the existing dependencies between data buffer accesses and the respective synchronization primitives, there are less obvious dependencies that should also be preserved to maintain determinism and avoid deadlocking code generation.

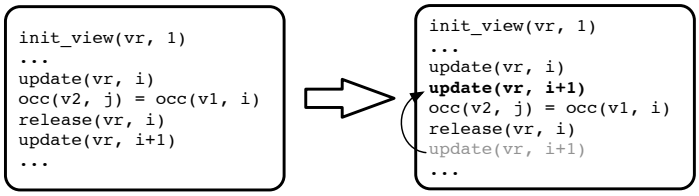


Figure 6.5: Code transformation ignoring initial specified horizon size.

To guarantee a minimum valid record buffer size, compilers rely on view horizons to define big enough buffers, accommodating all the algorithmic needs for the record connected processes (views). Moreover, compilers should guarantee that the sliding window size never exceeds the horizon size, i.e. the index difference of consecutive release and update primitive calls should never exceed the view horizon size.

The following two dependency equations guarantee it never occurs.

$$release(v, i_r)\ \delta\ update(v, i_u) \quad \textbf{if} \quad 0 < i_u - i_r \leq horizon(v)$$
$$commit(v, i_c)\ \delta\ stall(v, i_s) \quad \textbf{if} \quad 0 < i_s - i_c \leq horizon(v)$$

These equations represent the back edges between *release* and *update*, as well as *commit* and *stall* presented in Figure 6.4. These dependencies at first sight might seem counter intuitive considering the previous presented meaning of horizon. Instead, such dependencies exploit the fact that in order for the sliding window not to exceed the view horizon, the dependency sink (*updates*) must always have an index between the dependency source (*release*) current index (i_r) and within the range of horizon ($i_r + horizon$). An *update* out of these ranges is either irrelevant or too big for the expected view horizon size.

Figure 6.5 is a code transformation example not taking in consideration the previous specified dependencies and resulting in an invalid view initialization. As determinism is dependent on correct initialization of its views, such transformation is invalid. This is the case because the view is initialized with view horizon equal to 1. As an example, considering the insufficient horizon size after the transformation, record buffers can be allocated with insufficient size for this particular process. Partial redundancy elimination optimization, as later explained in the chapter, performs such code transformations, guaranteeing code correctness through the usage of these dependency equations.

Process self dependencies

Some processes might contain dependencies that by process code analysis are unpredictable, requiring a full application graph or higher abstraction level (PNG data structure) analysis in order to further analyze the process dependencies. Examples are processes containing a reader view transitively connected to at least one of its writer views.

Such scenarios are problematic considering that wider analysis are necessary in order to guarantee the operations non dependency. Without such analysis, one could incorrectly assume that unrelated view operations are independent and incorrectly optimize them. Nevertheless, such equations do not take in consideration any possible dependencies external to process code.

Figure 6.6 shows two application graphs with self dependent process instances and invalid transformation examples, ignoring such dependencies. Using only the previous presented

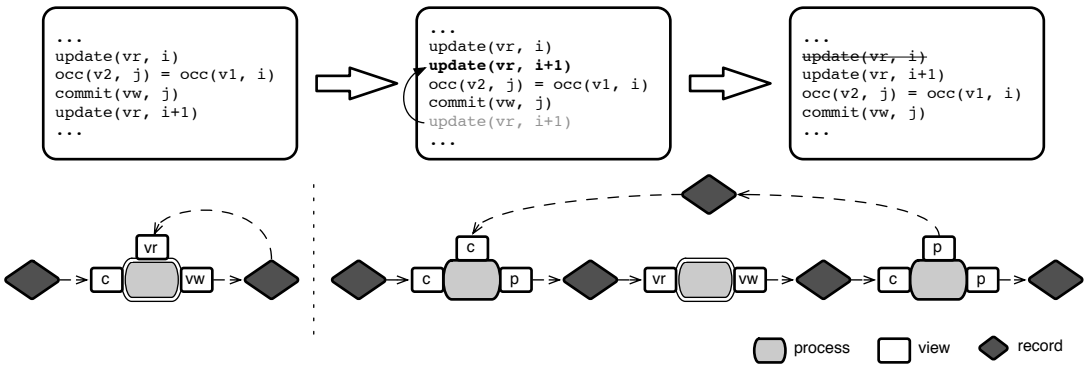


Figure 6.6: Possibly invalid code transformation relying only on the previously defined dependencies, as well as, two application graphs that can generate such type of dependency. The double-lined boxes represent the processes possibly containing the code in the invalid transformations above.

dependences, one might immediately consider to anticipate the latest update and possibly later remove one of the update operations thanks to the introduced redundancy. However, at a global application scope, such commit might be affecting data of a dependent record required by a succeeding update operation. Anticipating such update creates a deadlock since the dependent commit would not be executed before the update, making the update to block.

$$commit(v_1, i_c)\ \delta\ update(v_2, i_u) \quad \textbf{if} \quad v_1 \xrightarrow{+} v_2 \wedge i_c - i_u \leq \textbf{slack}\,(v_1 \xrightarrow{+} v_2)$$
$$release(v_1, i_r)\ \delta\ stall(v_2, i_r) \quad \textbf{if} \quad v_1 \xrightarrow{+} v_2 \wedge i_r - i_s \leq \textbf{slack}\,(v_1 \xrightarrow{+} v_2)$$

Moreover, not always a commit is dependent on a successor update, but only in cases where there is some dependency between its writer and reader views. In such cases, deeper analysis are required in order to further understand the dependency. Furthermore, those two operations are only dependent if there indexes, in the worst case, can generate a deadlock, i.e., are dependent.

To really determine it, it is necessary to compute the *slack* for such transitive connection (mentioned earlier in the chapter) and to verify if the difference between commit and update indexes (index $i_c - i_u$) is smaller or equal to the slack for the transitive connection dependency ($slack(v_1 \xrightarrow{+} v_2)$).

Primitives ordering and dependencies

Erbium primitives dependencies, as presented, are not only related to the primitives index provided in its argument but as well to the executing ordering of the dependent primitives. Let us consider a more relaxed dependency type or *index dependency* (δ_i) where only the primitive argument is verified.

Any two operations do not necessarily get dependent based on their execution proximity or ordering. Nevertheless, primitives ordering is a requirement for Erbium applications determinism. Moreover, determinism is a property of the ordering of two dependent primitives.

Consider **update** (v, i) δ_i **occ** (v, j), by being dependent it means that $0 \leq i - j < horizon(v)$. If *update* is a predecessor of *occ* then it is guaranteed that such index dependency ($delta_i$) is also a dependency (δ) guaranteeing a deterministic use of *occ* operation. On the other hand, if *update* is not a predecessor, then in order to guarantee determinism of this particular *occ*, it is necessary that some other *update*, let us say $update(v, k)$, has a dependency to the *occ* call and covers all the remaining control-flow paths to which the previous primitive did not.

$occ(v, j)$ is deterministic if, for all possible execution paths it exists a primitive a such that a δ **occ** (v, j), which by the dependencies presented can only be an *update* or a *stall*.

It is in the interest of Erbium IR to maintain determinism through its compilation flow. Nevertheless, its representation should never try to fix non deterministic applications. For that same reason, index dependencies do not take in consideration the primitives execution order. Let us consider an opposite example where *update* is a successor of *occ* primitive and the *occ* does not have deterministic semantics. In such a case, the compiler can trigger a warning to the user, signaling a non recommended usage of the primitives, however the application primitive ordering should never be interchanged in this case.

Determinism is maintained by guaranteeing that from the many dependency equations, say $a \, \delta \, b$, a is always executed before b. Not being so does not make these two primitives non dependent, but only dependent in a less expected manner (in a non determinate one), or only with an index dependency. The other dependency equations not involving the *occ* operation also require to occur in the same ordering, nevertheless such dependencies do not directly impact the *occ* determinism but rather violate the horizon size and other inter-process dependencies, possibly resulting in non deterministic executions in any case.

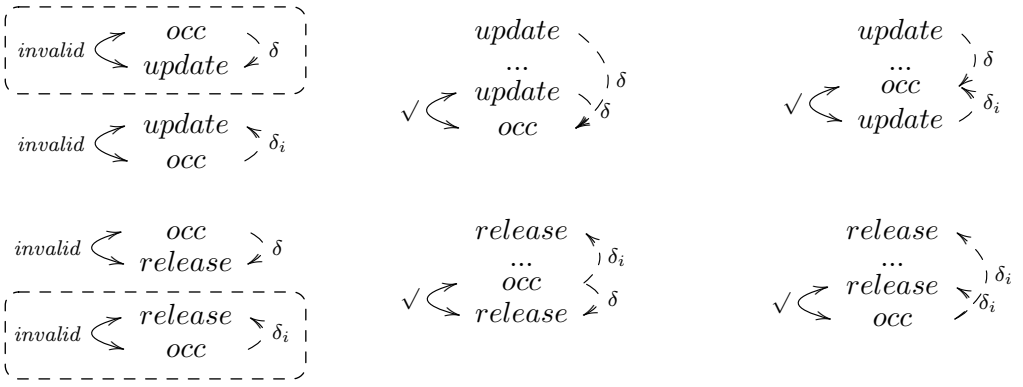
Swapping rules - control-flow and dependencies

The *occ* operations can now be classified as either being deterministic or non deterministic and their behaviour is solely the result of being dependent or only index dependent to the synchronization primitives. Any possible code transformation should never change the meaning of a specific *occ*, albeit non deterministic *occ*s have no predictable outcome. In any case, an application programmer could be doing it on purpose.

Many times, it is possible to interchange two dependent operations, without affecting the application semantics or behaviour of the particular primitive. Interchanging two dependent primitives is possible if a similar type primitive is executed is a predecessor of the dependency sink. For example, let us consider three primitives a, b, c such that $a \, \delta \, c$ and $b \, \delta \, c$, if a is a predecessor of b, then the primitives in the dependency $b \, \delta \, c$ can be interchanged without affecting the semantics of the primitive c.

The *update* and *release* primitives impact the positioning of the view sliding window and the data availability to its associated process. Once a particular type of dependent primitive is executed, the remaining dependent primitives have no semantical meaning. The following

diagram presents several examples of valid and invalid *occ* primitive swap operations.



The left most swap examples are between two dependent primitives where the *occ* has no dependency to a second primitive. In such a case and while preserving the behaviour of the *occ* both swaps are invalid, considering the swap will convert either a deterministic *occ* into a non deterministic one. Nevertheless, in cases where the compiler must enforce a deterministic behaviour the boxed examples are valid, since it makes the *occ*s deterministic.

The remaining examples are related to the swapping of relations when a predecessor primitive is also dependent on the *occ*, in such a case reordering the primitives is possible without changing the behaviour of the application. Any other Erbium primitive dependencies must also respect to these same swapping rules.

Most code motion transformations, in current generation compilers, occurs in few optimizations such as constant propagation and loop invariant code motion, many times implemented together in the partial redundancy elimination (PRE) optimization. Code motion of Erbium primitives does not occur through primitive swapping, as presented in this section, but rather with the insertion and redundancy detection of primitives, as PRE optimization does. A PRE extension to Erbium primitives is presented later in the chapter.

6.1.3 Non Erbium code dependencies

As explained in Chapter 4, during compilation flow the *occ* builtin calls are converted to direct memory buffer accesses, hiding the overhead of calling a function to retrieve the event index memory pointer. Moreover, all the synchronization builtin calls are patched with a new argument (the buffer pointer), making any synchronization call directly dependent to the full buffer memory. From this point on, the previous data dependencies no longer relate to *occ* primitives but rather to full buffers. This disables any future analysis based on view sliding windows or Erbium indexes.

A non Erbium code is not directly dependent on any synchronization primitive. However, previous converted *occ*s, now buffer accesses, are still related through its memory accesses. Since buffers are dependent on the synchronization primitives, this chain of dependencies guarantees that synchronizations are still protecting the buffer accesses, as well as other buffer dependent non Erbium code.

The finer dependency level is the less expressive in respective to the possible parallel optimizations, although perhaps it is the most important, considering the number of already existing optimizations in current generation of mainstream compilers.

At this level, buffer accesses get the chance to be further optimized together with all the non Erbium related code. Examples of such buffer related optimizations are: reducing the actual number of buffer accesses through register promotion, converting any intermediate buffer accesses into registers, or even vectorize the process loops if the process burst (data-sample) is big enough. Such optimizations occur through traditional optimizations, possible considering that the *occ* primitive is converted into direct buffer accesses, similar to array accesses. At this abstraction level, PRE no longer moves synchronization primitives with respect to buffer accesses, since the monotonic indexes are lost during the conversion of *occ*.

Furthermore, builtin helper functions for collecting the buffer memory have attribute *const* in order to minimize penalty of using such builtin operations. If that was not the case, every previous converted *occ* builtin would be surrounded with side-effect builtins (the buffer getter calls) which would obfuscate many optimizations on the memory accesses and its surrounding sequential code. These *const* builtin calls are hoisted out of the main computational loop and moved next to the view connection builtin call, where buffers are "defined". Please remember the previous presented conversion of *occ* builtin primitive, explained at Section (4.3.2), and demonstrated at Figure 4.11.

6.1.4 Summary of dependencies

$$
\begin{aligned}
update(v, i_u)\ \delta\ occ(v, i_o) \quad & \textbf{if} \quad 0 \leq i_u - i_o < horizon(v) \\
occ(v, i_o)\ \delta\ release(v, i_r) \quad & \textbf{if} \quad i_o \leq i_r \\
\\
stall(v, i_s)\ \delta\ occ(v, i_o) \quad & \textbf{if} \quad 0 \leq i_s - i_o < horizon(v) \\
occ(v, i_o)\ \delta\ commit(v, i_c) \quad & \textbf{if} \quad i_o \leq i_c \\
\\
occ(v_r, i_1)\ \delta\ occ(v_w, i_2) \quad & \textbf{if} \quad a \leftarrow occ(v_r, i_1) \wedge occ(v_w, i_2) \leftarrow b \wedge a\ \delta\ b \\
\\
release(v, i_r)\ \delta\ update(v, i_u) \quad & \textbf{if} \quad 0 < i_u - i_r \leq horizon(v) \\
commit(v, i_c)\ \delta\ stall(v, i_s) \quad & \textbf{if} \quad 0 < i_s - i_c \leq horizon(v) \\
\\
commit(v_1, i_c)\ \delta\ update(v_2, i_u) \quad & \textbf{if} \quad v_1 \xrightarrow{+} v_2 \wedge i_c - i_u \leq \textbf{slack}\,(v_1 \xrightarrow{+} v_2) \\
release(v_1, i_r)\ \delta\ stall(v_2, i_r) \quad & \textbf{if} \quad v_1 \xrightarrow{+} v_2 \wedge i_r - i_s \leq \textbf{slack}\,(v_1 \xrightarrow{+} v_2)
\end{aligned}
$$

6.2 Erbium compiler flow

The three explained abstraction levels define the three distinct levels of expressiveness of the Erbium language, each capable to perform different types of analysis and optimizations.

The highest level of abstraction presents to compilers the opportunity to optimize the application at an inter-process level, mostly using the provided high-level languages properties, collected through the language front-end lowering or source-to-source compiler. The collected information is not stored in the Erbium primitives, but rather in the PNG data structure which depends on the original language properties.

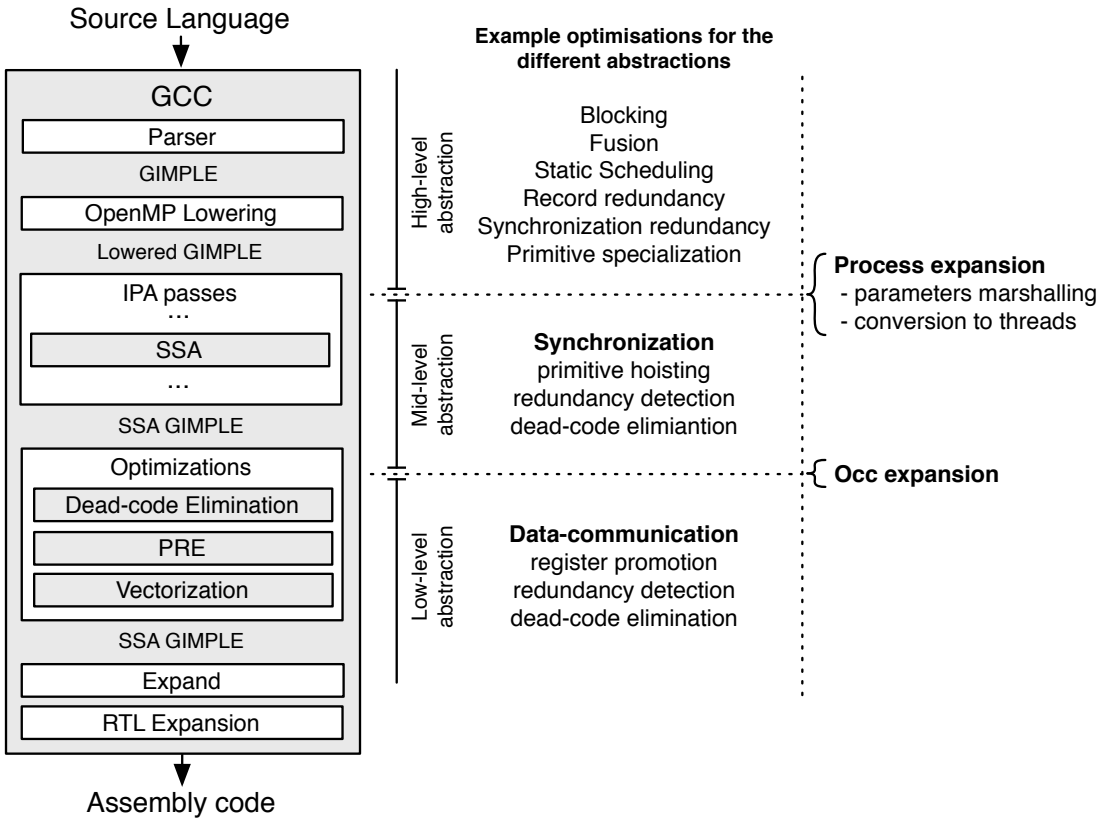


Figure 6.7: Erbium's abstraction levels integration in compilation flow, associated with its level optimizations.

This level of abstraction is available during the first passes of the compiler execution. In GCC it is during inter procedural analysis (IPA) passes, where function replication or IPA related optimizations are easily applied. Inter-process optimizations, although not only possible at this abstraction level, are highly recommended if no particular analysis of the process code are necessary, i.e., in case higher level language properties provide sufficient data for a correct decision. This is the case of SDF computational model languages, computing process fusion and blocking.

The following abstraction level refers to synchronization primitives optimizations. At this level, synchronizations and data communication (*occ*) primitives are still in builtin form and use monotonic indexes to refer to record events. Thanks to the dependencies presented in Section 6.1.2 and summarized in Section 6.1.4, it is possible to infer possible code motions by extending existing compiler optimizations such as partial redundancy elimination and dead-code elimination. Synchronization and data-communication code motion and dead-code elimination are important considering that both the high-level language code lowering and previous optimizations might introduce spurious or synchronization calls not optimally placed. This level of abstraction uses *occ* builtins to compare the primitives monotonic indexes and provides PRE pass with enough information to perform its analysis.

The lowest abstraction level is justified by the lowering of the *occ* primitives to direct array like buffer operations. At this level it is possible to detect and remove repeated buffer accesses, to promote the intermediate buffer accesses by registers, and still to perform any of

the general compiler optimizations as would occur in a sequential version of the same code, i.e. if no synchronization primitives were present. Although no new optimizations are presented at this level, this level provides to Erbium its most important optimizations considering the amount of traditional optimizations already available in mainstream compilers such as GCC.

Figure 6.7 is a GCC's compiler-flow diagram, presenting the positioning of the three abstraction levels, associated conversions and optimizations.

6.3 Example optimizations

The Erbium language has very low level parallel constructors, decoupling process creation, synchronization and data communication. Decoupled constructs make it extremely hard to analyze and eventually transform considering the uncorrelated communication and synchronization primitives. Nevertheless, such properties also allow finer optimizations that would not be possible with higher level constructs, fitting better the target architecture properties.

This section presents several Erbium related optimizations, which, although possible to implement in Erbium, were not further studied and are open for future research.

6.3.1 Process blocking

High-level programming languages hide parallel programming complexities though a set of constructs that implicitly expose parallelism and data communication. These languages programs have more general implementations, abstracting the programmers from all the complexities of the target architectures. These abstractions are then optimized during compilation, when further details on the target architecture are known.

Optimizations such as process blocking (not to be mistaken with thread blocking) are responsible for such adjustments and more precisely to adjust data communication and synchronization granularity to the target architecture memory size (caches or private memories), memory latency and synchronization overhead. In many cases, process blocking also enables loop vectorization, which greatly increases the process performance.

In the high-level optimizations perspective, blocking implies changing the process consumption and production burst typically associated with the language task definition. In Erbium and in the perspective of the process code, such optimizations resemble loop level optimizations such as loop tiling.

Figure 6.8 is a transformation example much similar to what an Erbium optimization could do to perform process blocking. In this optimization, we replicated the main loop of the process code and tiled the loop of one of the copies, creating an inner loop which iterates *BLOCKING_FACTOR* times. Executing the original or tiled loop version depends on the availability of sufficient data in the record, guaranteed by the builtin *view_enquire*.

As one can see, synchronization calls remain inside the loop. Current transformation already improves data locality, however as the synchronizations are repeatedly called for every loop iteration, synchronization overhead does not change. To reduce their overhead, synchronization calls must be hoisted outside of the new loop. Moreover and as blocking depends on the increase and validation of the new horizon sizes, the synchronization primitives hoist is left to the extended PRE optimization pass (later explained), which with the increased view horizon size, is able to hoist the synchronization primitives out of the newly generated loop.

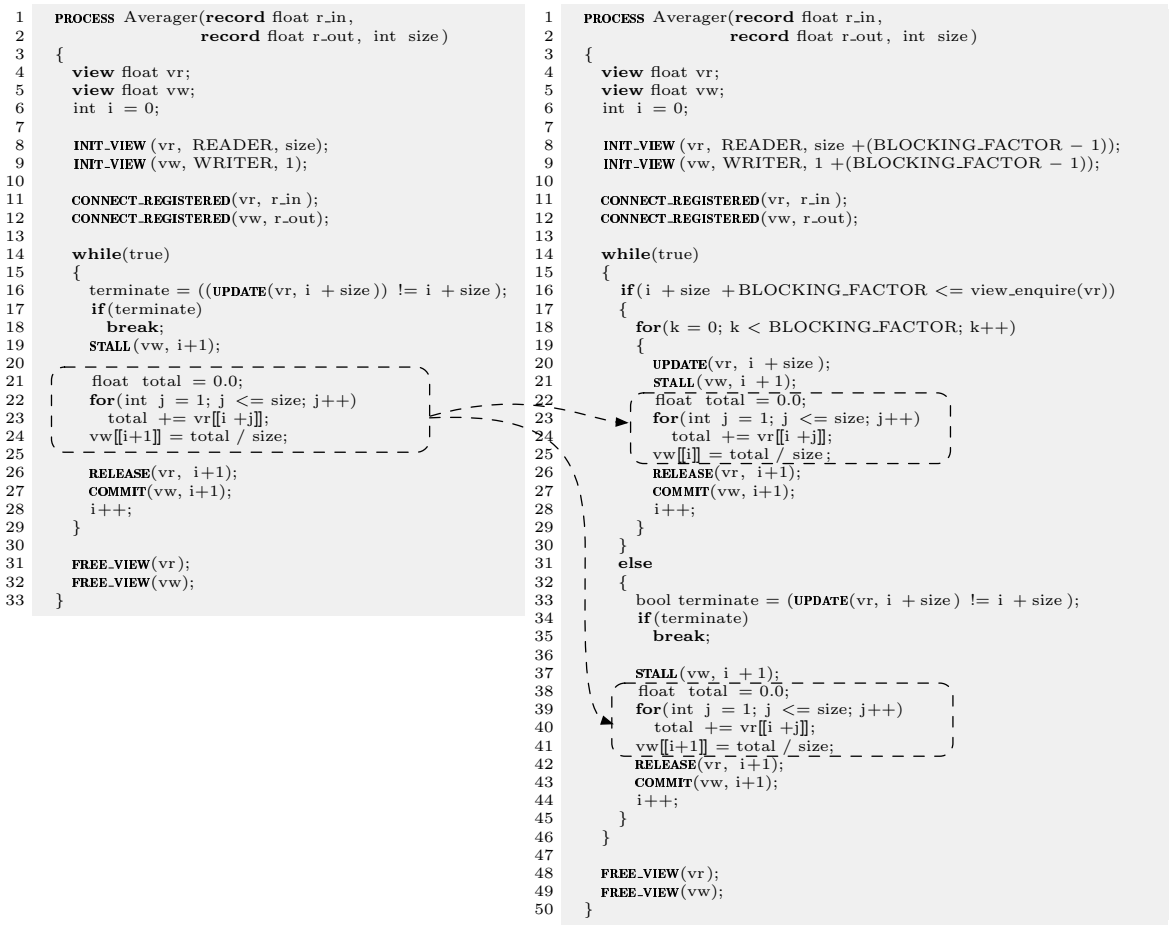


Figure 6.8: Blocking optimization applied to *Averager* process.

Figure 6.9 presents the full blocking code transformation after the synchronization hoist performed by PRE, considering the increased horizon size. In the example both versions of the loops are fused using a new temporary variable *tmp*, defining the amount of data elements synchronized at next loop iteration. This type of existing code analysis and transformations is one of the benefits of using such a high-ranked mainstream compiler such as GCC.

Many times blocking is associated with code vectorization. As multiple data elements are combined in a single synchronization, it might become possible to vectorize the data-elements computation, reducing the execution time significantly.

Moreover, as process blocking implies an increase in the view horizon sizes, the compiler must verify this transformation in a global application scope against possible deadlocks. The verification is based on the PNG collected view burst, slack and buffer sizes information.

The code duplication is necessary to preserve original process behaviour (non tiled code), preserving the original behaviour in case no sufficient data is available to consume, as probably is the case on termination where no new data is certainly not committed (after the records get zombified).

The blocking optimization support implies the implementation of a new builtin primitive which allows to peek the latest events committed or released by the respective view record (*view_enquire*). Such type of primitive, although not presented and recommended as a general

```
1  PROCESS Averager(record float r_in,
2                   record float r_out, int size)
3  {
4    view float vr;
5    view float vw;
6    int i = 0, tmp;
7
8    INIT_VIEW(vr, READER, size +(BLOCKING_FACTOR - 1));
9    INIT_VIEW(vw, WRITER, 1 +(BLOCKING_FACTOR - 1));
10
11   CONNECT_REGISTERED(vr, r_in);
12   CONNECT_REGISTERED(vw, r_out);
13
14   while(true)
15   {
16     if(i + size +BLOCKING_FACTOR <= view_enquire(vr))
17       tmp = BLOCKING_FACTOR;
18     else
19       tmp = 1;
20
21     bool terminate = UPDATE(vr, i +tmp +size) != i +tmp +size;
22     if(terminate)
23       break;
24
25     STALL(vw, i +tmp);
26     for(k = 0; k < tmp; k++)
27     {
28       float total = 0.0;
29       for(int j = 1; j <= size; j++)
30         total += vr[[i +k +j]];
31       vw[[i +k]] = total / size;
32       i++;
33     }
34     RELEASE(vr, i +tmp);
35     COMMIT(vw, i +tmp);
36   }
37
38   FREE_VIEW(vr);
39   FREE_VIEW(vw);
40 }
```

Figure 6.9: Process blocking after partial redundancy elimination.

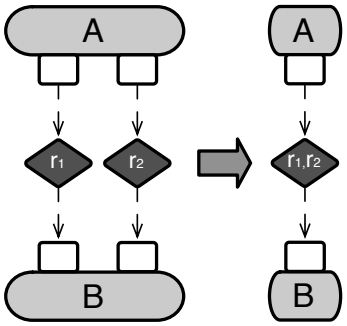
language primitive considering its extra expressiveness, possibly leading to non-determinate applications, can be safely inserted by compilers, considering it is never used to increase the application semantics but only as an optimization primitive. Moreover, the blocked optimized processes are only specializations of the original code, preserving the process original determinism.

6.3.2 Fusion

Highly partitioned parallel applications tend to waste too much time switching between parallel threads. In applications where the number of process instances is higher then the available parallel processors, coarsening parallelism, by fusing multiple processes, significantly reduces the number of thread switches.

However, the process fusion requires to verify and match the production and consumption rate of the fusing process. For example, in order to fuse two processes it is necessary that both processes are directly connected and have multiple production and consumption rates. When the rates do not match the processes can possibly be adapted using the previously explained blocking transformations, balancing the processes burst rates.

Process fusion, like blocking, implies the identification of the task single entry single exit (SESE) code region. Languages based on SDF and CSDF are composed of SESE tasks, simplifying process fusion code generation.




```
int s1, r1
float r2;
#pragma task input(s1) \
            output(r1, r2)
  {
    r1 = f(s1);
    r2 = g(s1);
  }
#pragma task input(r1, r2) \
            output(s2)
  {
    s2 = h(r1, r2);
  }
```

```
typedef struct r1r2 {
  int r1;
  float r2;
}
int s1, s2;
struct r1r2_struct r1r2;

#pragma task input(s1) \
            output(r1, r2)
  {
    r1r2.r1 = f(s1);
    r1r2.r2 = g(s1);
  }
#pragma task input(r1, r2) \
            output(s2)
  {
    s2 = h(r1r2.r1, r1r2.r2);
  }
```

Figure 6.10: SOMP record fusion example done with SOMP code.

6.3.3 Record fusion

The code conversion of high-level languages into Erbium might generate suboptimal code, creating too many records for the existing data communications. For example, in Chapter 5, the presented code conversion created a single record for each stream variable, resulting in the creation of multiple records in cases where more then one stream variable were used in the connection of the same tasks. Moreover, after performing record fusion, it is also easier to detect and perform the previously presented transformation, record fusion.

If a process A connects, as writer, to record r_1 and r_2, and process B connects as reader to both these records and no other processes connect to these records, it could be possible to replace those records by a single record whose type is a data structure containing both the record types.

More generally, if a set of records are always accessed simultaneously with the same type of connection and at the same rate (inside each process), those records can be merged and replaced by a single one. Multiple views connecting to the previous record are substituted by a single view.

In order to decide on the merge of a set of records, it is sufficient that:

- if a process connects to one of the records in the set, it should also connect to the remaining of the records,
- all the process connections should be of the same type (reader or writer),
- for every individual process, the index of the different associated view synchronization calls should be the same,
- similar synchronization calls should happen in same control-flow points.

Once all these conditions are valid, the set of records can be combined in a single one by:

- combining all the different record types into a single data structure,
- per process, merging all the views and synchronization calls,
- traversing all previous view *occ* primitives substituting by the newly view, adding the respective data structure offset.

As one can imagine, validating for record fusion relying on Erbium code analysis is very hard and sometimes even impossible. However, if we consider the available information provided by high-level languages, such analysis are unnecessary. Figure 6.10 presents the record fusion for original (left) and transformed (right) SOMP code examples. The Erbium

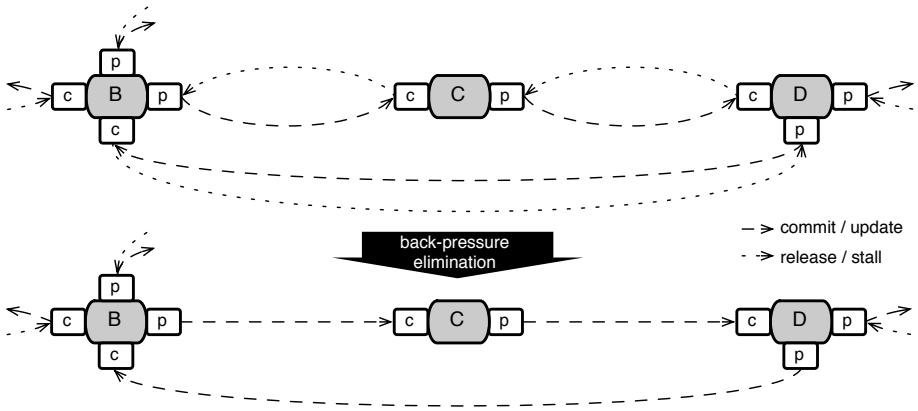


Figure 6.11: Back-pressure elimination transformation example.

transformation should perform the same as the difference between both these SOMP code examples after its conversion.

The record fusion improves application performance through the elimination of redundant synchronizations (reducing the number of views in the process) and improves data locality associated to the merge of the record types into a data structure.

6.3.4 Back-pressure elimination

Other form of synchronization redundancy can also be detected in back pressure primitives in cyclic process communications. Figure 6.11 is a graphical representation of such transformation.

The Erbium back pressure synchronization introduces as much overhead as data related synchronization primitives. Albeit delaying resources freeing, back-pressure elimination reduces synchronization overhead in half,

To get perform improvements the application requires large enough record buffers — big enough to allow a full communication cycle before buffer saturation. Moreover, the processes require to increase the horizon size, making sure sufficient space is available in their records, i.e., as processes will not contain calls to the *stall* primitive, the resources availability should be verified through *update*, checking for the commit of the last process in the cycle (its cycle predecessor).

Such optimization is highly useful for distribute memory architectures, such as the Cell BE and its SPUs, where processors are interconnected in a multichannel ring. The unidirectional traffic significantly reduces the number of conflicting communications and so, its DMA transfer overhead and the overall application execution time.

Non cyclic applications can also take advantage of this transformation by introducing a "fake" back connection (record), creating and closing the "fake" cycle.

6.3.5 Static primitive specialization

Chapter 3 presented a runtime implementation of the Erbium primitives supporting multiple producers and consumers.

Many applications do not make use of the multiple record connectivity. Nevertheless, runtime implementations contemplate many of the complexities introduced by multiple record connectivity. Please refer to Figures 2.2 and 2.7, presented in Chapter 2, for a comparison of the dependencies associated with single vs. multiple connectivity records, providing a glimpse of the increased complexity.

In cases where static prediction is possible, and once it is detected that a specific record is never connected by more than one reader and writer view, the involved processes can instead call specific versions of the synchronization primitives, not performing all the checks and computations required by multiple producer and consumer synchronizations. The minimum computation required for multiple producers and consumers implies very often traversals to the list of views. Moreover, if the minimum computation is not necessary, the record does not requires to implement a list of views, with all of its concurrency protections, but instead a pair of views, without any protections, since it is guaranteed that only a single thread writes to the record.

6.3.6 Redundant synchronization calls

The implementation of Erbium synchronizations is expensive, depending on the target architecture and its runtime library support.

Like dead-code elimination, redundancy detection can, at first sight, seem irrelevant considering that no programmer or language conversion should generate redundant code. Nevertheless, optimizations very often introduce redundancies. A simple example of these redundancies are the consecutive calls to the same synchronization primitive, such as *update*. Multiple similar and consecutive synchronization primitives for the same view can always be substituted by a single one with the maximum index for all the calls index arguments.

In this chapter, we will present an extension to partial redundancy elimination (PRE) where synchronizations are, in some particular cases, possibly moved and eliminated, removing any redundant or spurious synchronization calls by relying on the dependencies graphically presented in Figure 6.4 and summarized in Section 6.1.4.

Similarly to PRE, dead-code elimination can also predict when synchronizations are not needed. In Erbium this is the case if a particular range of events is never used or does not contribute to the application output, i.e., if the data read from the view buffer is never used, or never displayed. In these cases, the *occ*s are flagged as dead code and the synchronization primitives, when only guaranteeing the determinism of the *occ*s are also considered as dead code.

6.3.7 Optimum synchronization placement

Having PRE to perform code motion does not necessarily mean an improved placement of synchronization primitives. PRE requires a set of heuristics specifically design for Erbium synchronization primitives. For example, the *update* primitive should be executed as late as possible. Let us consider an *update* being called before some independent slow function call. By preserving the call ordering *update* call has far more probability to block than if both calls get reordered.

However, let us assume *update* is inside a loop and, based on the view horizon size, it could cover several executions of this loop (like what would occur with the process blocking optimization). Should the *update* call be hoisted before the loop? Doing so enforces process

synchronization to happen before the slow function executes. Not always it is easy to deduce which transformations should be done or not. Many times, code simplification or redundancy removals do not translate in speedups. The *commit* and *release* primitives, contrarily to *update* and *stall*, while respecting its dependencies, should execute as soon as possible, as they do not block and announce the record progress.

As mentioned in the two previous examples, it is not always easy to predict a best placement for the synchronization primitives. These transformations must be supported by cost prediction algorithms, deciding between the different transformation opportunities.

PRE as later presented in the chapter does not tries to optimize code placement, but instead present how PRE is extended to support the Erbium primitive calls, performing code motion and redundancy detection. Later research work is needed to define and implement the heuristics to optimize the Erbium primitives placement.

6.4 Synchronization PRE

Although Erbium intermediate representation as presented in Chapter 4 does not break traditional PRE, by itself the intermediate representation does not support any code motion of Erbium primitives, i.e., does not allow PRE to improve Erbium code. Erbium code motion requires PRE code analysis and transformation heuristics to support its builtins, more precisely its primitive relations based on its monotonic indexes, as expressed in the dependency equations summarized in Section 6.1.4.

The PRE code analysis implies the creation of availability and anticipability sets. For PRE pass, availability sets represent the statements forward movable in the control-flow, computed for each program point. The anticipability represents the statements movable backwards in the control-flow at given program-point. The PRE heuristics traverse both the sets and, per program-point, validate all the possible code insertions, inserting new statements which make previous original statements redundant and eventually removes them. The insertion and deletion of statements is an iterative process, where previous PRE transformations reveal new redundant statements.

Extending PRE implies extending the availability and anticipability sets taking into consideration the Erbium primitives semantics.

Availability is the set of all expressions available at a given program point. A given expression $(a + b)$ is available at a program point or statement s if, on every path from the beginning of the control-flow graph to s, the expression $(a + b)$ is computed at least once and none of the variables used in the expression (a and b) has been redefined since the latest computed expression.

Anticipability is the set of all expression anticipable at a given program point. Similar to the previous example, an expression $(a + b)$ is anticipable at a given program point s if, every path from s to the end of the control-flow graph, $(a + b)$ is computed and none of the variables a and b is changed before the first computation of $(a + b)$.

A different definition of availability and anticipability is decomposed in *in* and *out* sets, representing availability or anticipability as widely explained in compilation literature [10].

Statement s	Availability		Anticipability	
	gen$[s]$	**kill**$[s]$	**gen**$[s]$	**kill**$[s]$
$t \leftarrow b \oplus c$	$\{b \oplus c\} - kill[s]$	expressions with t	$\{b, c\} - kill[s]$	$\{t\}$
$t \leftarrow M$[b]	$\{M[b]\} - kill[s]$	expressions with t	$\{b\} - kill[s]$	$\{t\}$
M[a] $\leftarrow b$	$\{\}$	expr. of form $M[x]$	$\{b\}$	$\{\}$
if $a > b$ *goto* ...	$\{\}$	$\{\}$	$\{a, b\}$	$\{\}$
goto L	$\{\}$	$\{\}$	$\{\}$	$\{\}$
L :	$\{\}$	$\{\}$	$\{\}$	$\{\}$
$f(a_1, \ldots, a_n)$	$\{\}$	expr. of form $M[x]$	$\{a_1, \ldots, a_n\}$	$\{\}$
$t \leftarrow f(a_1, \ldots, a_n)$	$\{\}$	expressions with t, and expr. of form $M[x]$	$\{a_1, \ldots, a_n\} - kill[s]$	$\{t\}$

Table 6.1: *Gen* and *kill* sets for sequential languages such as C.

Availability and anticipability is computed iteratively with the following two equations:

$$\textbf{Availability} \qquad \begin{aligned} in[n] &= \bigcap_{p \in Pred(n)} out[p] \\ out[n] &= gen[n] \cup (in[n] - kill[n]) \end{aligned} \tag{6.2}$$

$$\textbf{Anticipability} \qquad \begin{aligned} in[n] &= gen[n] \cup (out[n] - kill[n]) \\ out[n] &= \bigcap_{p \in Succ(n)} in[p] \end{aligned} \tag{6.3}$$

As one might realize, such equations operate recursively, computing *in* and *out* of the successor statement in case of anticipability, or the predecessor if availability. *In* always refers to the immediate predecessor and *out* to the immediate successor program points[1]. Availability is computed based on a forward control-flow order statements traversal while anticipability is computed in a backwards order.

The previous definitions use the still undefined $gen[n]$ and $kill[n]$ sets, representing the newly generated expressions and the killed or invalidated expressions for the specific statement n. Common sequential languages operations define *gen* and *kill* sets based on the rules of Table 6.1 as presented by Appel [10] in pages 390-391.

GCC's PRE, or more precisely GVNPRE [83, 84], is an integration of Global Value Numbering and PRE optimizations, allowing to exploit both GVN and PRE optimizations simultaneously, while exploiting GCC's SSA GIMPLE intermediate representation. Previous most relevant contributions to PRE are Lazy Code Motion (LCM) introduced by Knoop et al. [47] and SSA-PRE by Kennedy et al. [45]. LCM is expression-based and thus lexical. On the other hand SSA-PRE, although the basis of GVNPRE, requires to change SSA form introducing a new type of node (Φ, not the same as the SSA ϕ node). GVNPRE does not suffer from the same caveat and is designed not to change original SSA form.

GVNPRE represents availability and anticipability sets using bitmap data structures. Each bitmap position (a bit) specifies the existence of a particular value in the available or anticipable set. The bitmap related value positioning is defined through the value numbering

[1]Program-points always refer to a code placement between existing statements.

assigned during the availability computation traversal. Value numbering identifies similar value expressions with the same number identifier. The bitmap data structure simplifies unions, intersections and subtractions set computations, required by the availability and anticipability equations previously given (Equations 6.2 and 6.3).

6.4.1 Erbium's availability and anticipability

Availability and anticipability sets define which operations are downwards or upwards safely movable without affecting program semantics. *Gen* and *kill* sets, depending on their actual usage, are more primitive operations by specifying for a single statement which values (expressions, function calls or variables) can be generated or destroyed (killed) in the availability or anticipability sets.

In the Erbium primitives case, the dependency equations already define the Erbium primitives expectations. Moreover, *gen* and *kill* sets, for both availability and anticipability, can be defined using these dependencies:

For $\mathcal{S}$ being the set of all possible Erbium primitive calls.

Availability

$$\begin{aligned} gen[s] &= \{n \in \mathcal{S} : s \ \delta \ n\} \\ kill[s] &= \{n \in \mathcal{S} : n \ \delta \ s\} \end{aligned} \tag{6.4}$$

Anticipability

$$\begin{aligned} gen[s] &= \{n \in \mathcal{S} : n \ \delta \ s\} \\ kill[s] &= \{n \in \mathcal{S} : s \ \delta \ n\} \end{aligned} \tag{6.5}$$

Notice that the definitions for *gen* and *kill* equations are reversed when used either for availability or anticipability computation.

Consider the statement $occ(v, i)$, used in the context of a reader view, as an example for the computation of both *gen* and *kill* sets. The *gen* and *kill* are defined based on the universe of all the dependents for the particular statement s.

The *occ* primitive, in the context of a reader view, is associated with two other primitives, through the following dependencies, presented in Section 6.1.2 and the diagram presented in Figure 6.4:

$$update(v, i_u) \ \delta \ occ(v, i_o) \quad \textbf{if} \quad 0 \leq i_u - i_o < horizon(v) \tag{6.6}$$

$$occ(v, i_o) \ \delta \ release(v, i_r) \quad \textbf{if} \quad i_o \leq i_r \tag{6.7}$$

In the context of availability both *gen* and *kill* are computed through the expansion of the presented definitions.

$$\begin{aligned} gen[occ(v,i)] &= \{n \in \mathcal{S} : occ(v,i)\ \delta\ n\} \\ &= \{release(v,j) : i \leq j\} \qquad \textit{(by 6.7)} \\ &= \{release(v,j) : j \geq i\} \end{aligned}$$

$$\begin{aligned} kill[occ(v,i)] &= \{n \in \mathcal{S} : n\ \delta\ occ(v,i)\} \\ &= \{update(v,j) : 0 \leq j - i < hor(v)\} \qquad \textit{(by 6.6)} \\ &= \{update(v,j) : 0 \leq j - i \wedge j - i < hor(v)\} \\ &= \{update(v,j) : i \leq j \wedge j < i + hor(v)\} \\ &= \{update(v,j) : i \leq j < i + hor(v)\} \end{aligned}$$

By expanding each of the definitions through the various Erbium primitives, one can infer the *gen* and *kill* sets as presented in Table 6.2 for availability and Table 6.3 for anticipability.

In any case, not all of the presented dependency equations are used as *gen* definitions, more precisely the dependencies *commit* δ *update* and *release* δ *stall* which are inter-process dependencies. This is the case, because these dependencies are only necessary to guarantee deadlock free code motions in respect to the cyclic dependencies between processes, and not to anticipate new redundancies based on inter-process information, i.e., based on some other processes code.

The dependencies presented in Figure 6.4 define the chain of dependencies for the intra-process primitives, based on the view sliding window size (horizon). By analyzing the synchronization dependencies inside the process (intra-process), one can expect a maximum usage of the view horizon, i.e., PRE maximizes the number of events per synchronization primitive execution, as is further demonstrated. However, as these dependencies do not take into consideration process communication, it is necessary to limit the code motion possibly by the intra-process dependencies alone. For that matter, the *kill* definition should take into consideration the process inter-process dependencies (dependencies between processes) and disable any possibly invalid code motion, and more precisely avoid any PRE introduced deadlocks.

Tables 6.2 and 6.3 take inter-process dependencies in consideration only for the *kill* definition columns — more precisely, *commit* δ *update* and *release* δ *stall*.

$$release(v,i_r)\ \delta\ update(v,i_u) \quad \textbf{if} \quad 0 < i_u - i_r \leq horizon(v) \tag{6.8}$$

$$commit(v_1,i_c)\ \delta\ update(v_2,i_u) \quad \textbf{if} \quad v_1 \xrightarrow{+} v_2 \wedge i_c - i_u \leq \textbf{slack}\,(v_1 \xrightarrow{+} v_2) \tag{6.9}$$

Statement s	**gen**$[s]$	**kill**$[s]$
$u : update(v, i_u)$	$\{occ(v, j) : i_u - hor(v) < j \leq i_u\}$	$\{release(v, k) : i_u - hor(v) \leq k < i_u\} \cup$ $\{c : commit(v_1, l) : l \leq i_u + \textbf{slack}\,(v_1 \xrightarrow{+} v)\}$
$release(v, i_r)$	$\{update(v, j) : i_r < j \leq i_r + hor(v)\}$	$\{occ(v, k) : k \leq i_r\}$
$occ(v, i_o)$ read	$\{release(v, j) : j \geq i_o\}$	$\{update(v, k) : i_o \leq k < i_o + hor(v)\}$
$occ(v, i_o)$ write	$\{commit(v, j) : j \geq i_o\}$	$\{stall(v, k) : i_o \leq k < i_o + hor(v)\}$
$s : stall(v, i_s)$	$\{occ(v, j) : i_s - hor(v) < j \leq i_s\}$	$\{commit(v, k) : i_s - hor(v) \leq k < i_s\} \cup$ $\{c : release(v_1, l) : l \leq i_s + \textbf{slack}\,(v_1 \xrightarrow{+} v)\}$
$commit(v, i_c)$	$\{update(v, j) : i_c < j \leq i_c + hor(v)\}$	$\{occ(v, k) : k \leq i_c\}$
$connect(v, r)$	$\{update(v, i) : 0 < i < enquire(r) + hor(v))\}$ $\{stall(v, i) : 0 < i < enquire(r) + hor(v))\}$	Any primitives using v
$free(v)$	$\{\}$	Any primitives using v

Table 6.2: *Gen* and *kill* sets for Erbium primitives availability, generated thanks to Equations 6.4.

The following example expansion refers to the availability *kill* definition for $commit(v, i)$ taking in consideration the previous presented inter-process dependencies copied to Equations 6.8 and 6.9:

$$
\begin{aligned}
kill[update(v,i)] &= \{n \in \mathcal{S} : n\ \delta\ update(v,i)\} \\
&= \{release(v,j) \in \mathcal{S} : 0 < i - j \leq horizon(v)\} \qquad \textit{(by 6.8)} \\
&\quad \cup \{commit(v_1,k) \in \mathcal{S} : v_1 \xrightarrow{+} v \wedge k - i \leq \textbf{slack}\,(v_1 \xrightarrow{+} v)\} \quad \textit{(by 6.9)} \\
&= \{release(v,j) \in \mathcal{S} : 0 < i - j \wedge i - j \leq horizon(v)\} \\
&\quad \cup \{commit(v_1,k) \in \mathcal{S} : v_1 \xrightarrow{+} v \wedge k \leq i + \textbf{slack}\,(v_1 \xrightarrow{+} v)\} \\
&= \{release(v,j) \in \mathcal{S} : j < i \wedge i - horizon(v) \leq j\} \\
&\quad \cup \{commit(v_1,k) \in \mathcal{S} : v_1 \xrightarrow{+} v \wedge k \leq i + \textbf{slack}\,(v_1 \xrightarrow{+} v)\} \\
&= \{release(v,j) \in \mathcal{S} : i - horizon(v) \leq j < i\} \\
&\quad \cup \{commit(v_1,k) \in \mathcal{S} : v_1 \xrightarrow{+} v \wedge k \leq i + \textbf{slack}\,(v_1 \xrightarrow{+} v)\}
\end{aligned}
$$

The anticipability *gen* and *kill* definitions are expanded in the same manner.

Bitmap data structure extension

As presented in Tables 6.2 and 6.3 a single Erbium builtin statement generates and kills multiple primitives expressed by a range based on the statement index argument and view horizon. Expressing ranges is impossible using bitmaps, considering the infinite number of required bits to represent all the distinct primitive calls and its monotonic indexes arguments. Moreover, it would be meaningless since similar (same type and same view) primitives would

Statement s	**gen**$[s]$	**kill**$[s]$
$update(v, i_u)$	$\{v, i_u, release(v, j)) : i_u - h(v) \leq j < i_u\}$	$\{occ(v, k) : i_u - h(v) < k \leq i_u\}$
$r : release(v, i_r)$	$\{v, i_r, occ(v, j) : j \leq i_r\}$	$\{update(v, k) : i_r < k \leq i_r + h(v)\}$ $\{s : stall(v_1, l) : l \geq i_r -$ **slack** $(v \xrightarrow{+} v_1)\}$
$occ(v, i_o)$ read	$\{v, i_o, update(v, j) : i_o \leq j < i_o + h(v)\}$	$\{release(v, k) : k \geq i_o\}$
$occ(v, i_o)$ write	$\{v, i_o, stall(v, i_o) : i_o \leq j < i_o + h(v)\}$	$\{commit(v, k) : k \geq i_o\}$
$stall(v, i_s)$	$\{v, i_s, commit(v, j)) : i_s - h(v) \leq j < i_s\}$	$\{occ(v, k) : ki_s - h(v) < k \leq i_s\}$
$c : commit(v, i_c)$	$\{v, i_c, occ(v, j) : j \leq i_c\}$	$\{update(v, k) : i_c < k \leq i_c + h(v)\}$ $\{u : update(v_1, l) : l \geq i_c -$ **slack** $(v \xrightarrow{+} v_1)\}$
$connect(v, r)$	$\{v, r\}$	Any primitives using v
$free(v)$	$\{v, release(v, j) : 0 < j < \infty\}$ $\{v, commit(v, j) : 0 < j < \infty\}$	Any primitives using v

Table 6.3: *Gen* and *kill* sets for Erbium primitives anticipability, generated thanks to Equations 6.5.

be classified as different, i.e., every primitive would be different independently of its type and arguments.

To express the Erbium primitives in sets, it is necessary to adopt a more hybrid approach where the Erbium primitives are both classified and associated with a bitmap position and a range of index events. Each different type of operation (*update*, *release*, etc.) and the call first argument (the view) must be identified with a unique value number. Alias analysis has an important responsibility in the value numbering, considering that multiple pointers to the same view must be assigned the same value. The value numbering does the initial distinction of the possible redundant primitives, providing to similar Erbium operations the same value identifier, used later as the bitmap position.

Erbium primitives with same value number are not necessarily redundant, as is the case with traditional expressions. Redundancy of Erbium primitives is detected based on not only the primitive type (primitive call and view argument) but also based in its index argument, as previously expressed in the dependency equations. Considering the semantical differences between traditional expression redundancies and Erbium's, by itself the bitmap data structure is not sufficient to represent the extended sets. Moreover, in case of an Erbium primitive, each bitmap position is also associated with a tuple of index expressions, representing the range of available or anticipable primitive calls. Ranges are written with the expressions representing the limits presented in Tables 6.2 and 6.3 for both *gen* and *kill* sets depending on the primitive type, later combined in availability and anticipability computation. By still using a bitmap value for every combination of view and primitive type, lookup access times to both availability and anticipability are improved in cases where the primitive type is not present in the availability or anticipability sets.

Ranges

The usage of this new data structure to represent availability and anticipability is restricted to its capability to support the operations presented at availability (6.2) and anticipability (6.3) equations, more precisely: union, intersection and subtraction operations.

Such operations must be implemented in the same manner as traditional union, intersection and subtraction of set operations, although operating in a tuple of expressions representing a range.

By analyzing the range operations meaning, one can correctly presume a pair of tuples is not sufficient to cover the possible domain for the required operations. As an example, when two ranges A and B are disjoint $A \cup B$ cannot always be expressed with a single range. Similarly subtracting two ranges $(A - B)$ might also produce the same effect if A has different bounds and contains B. Nevertheless, as such ranges are only used in the context of Erbium primitive indexes, disjoint ranges are only the result of non dependent primitives. Moreover, when any of these operations produces more then one range, the result is always the continuous range with the highest values.

Index comparison

Availability and anticipability sets computation require the compiler to compare index arguments from the Erbium primitive calls, i.e., the *gen* and *kill* generated ranges are combined through union, intersection and subtraction operations, iteratively defining the availability and anticipability sets.

GCC allows to statically compare variables through scalar evolution analysis. In many cases, it is impossible to predict the evolution of such expressions considering that variables can be unpredictable thanks to complex code control-flow. In such cases, computing availability and anticipability for a specific Erbium primitive is impossible. However, for the many times, the index arguments are based on induction variables in organized loop hierarchies where scalar evolution analysis is available. For example, consider the operation $(\{update(v, A)\} - \{update(v, B)\})$, as it could happen in the availability computation (see Equation 6.2), and assuming the ranges A and B are incomparable, the operation result should be pessimistic, assuming $A \subseteq B$ and returning an empty set. In other words, when the comparison is not possible, the compiler should always act pessimistically and assume for the always correct case.

The scalar evolution analysis exploits SSA form and provides compiler optimizations with the ability to query the variable evolution in the context of precise program points. The outcome of scalar evolution is a polynomial representation for the variable evolution, where each variable is associated to number of iterations of each loop. This representation is named chains of recurrences (CHREC) [69, 70]. The chain of recurrences is the representation expressing the evolution of induction variables through loops. It is composed of an initial value, the operation occurring in every loop iteration and the loop stride. As the name suggests, several of these representations can be chained for multiple loop hierarchies.

A simple approach to index variable comparison is to independently compare the chain or recurrences components. If two chain of recurrences for the same program point contain the same operation and stride elements, both variables can be easily compared if their initial value is constant. If the initial values are other CHREC expressions, meaning the variable is defined outside another loop and, for that matter, it is related to at least another loop, the comparison should continue iteratively until the constant value is reached. When two CHREC representations have different operation or stride, those must be considered as incomparable. There should certainly be other more precise ways to compare CHREC expressions.

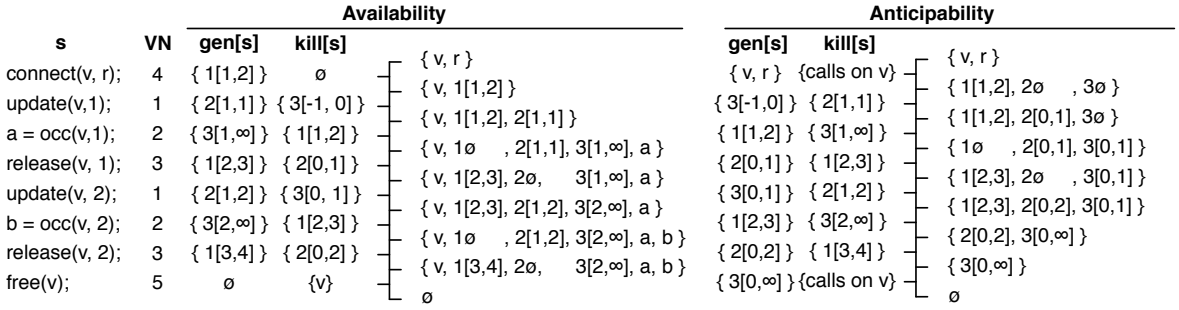


Figure 6.12: Availability and anticipability computation for a simple and constant index process. In this example the view horizon is 2 (not defined in the code) however the initial burst is 1.

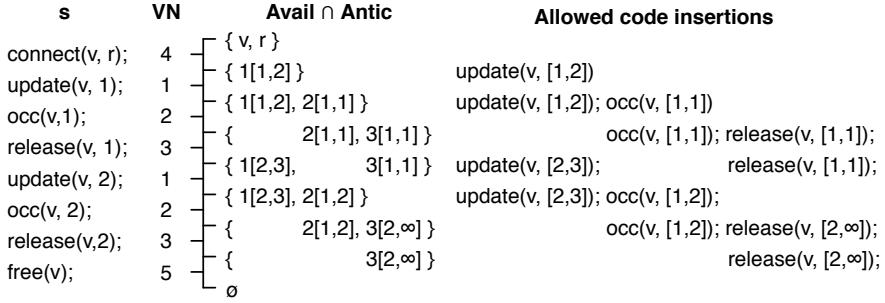


Figure 6.13: Intersection of availability and anticipability from the Figure 6.12. Allows to identify the semantically safe primitives at each program point. In this particular case it allows to predict that the second *update* can be safely anticipated thanks to the view horizon size of 2.

6.4.2 Availability and Anticipability computation example

Figure 6.12 is a example of *gen* and *kill* sets for a very simple single basic block application. The application is presented in the left-most column. The less relevant yet necessary statements such as record/view allocation and initialization are omitted from the code example considering its smaller interest in the availability and anticipability computation. Moreover, please take into consideration that the view horizon for v for this particular example is 2 (*view_init* is one of the omitted primitives).

Much like what happens currently in GCC, each primitive call is assigned with an unique value number (VN) for each primitive type and view argument tuple. The sets are defined with the notation $VN[begin, end]$, being *begin* and *end* the limit expressions for the range data structures. In the example, all primitives use constant index values, simplifying the resulting range limits and making them constants.

Gen and *kill* sets are merely the application of the rules presented in Tables 6.2 and 6.3. The right-most columns for each section are the availability and anticipability computations for the specific program point, as specified in the Equations 6.2 and 6.3. Availability is calculated based on a forward traverse of the statements, while anticipability is a reverse traversal, i.e., from the last statement to the first one.

As previously mentioned, the most simple code motion or redundancy insertion verification is to compute the intersection of both availability and anticipability sets. Any statement present in an availability and anticipability intersection can safely be inserted in that particular program point, probably generating redundancies and eventually allowing PRE to remove

the now redundant primitives. Figure 6.13 is the example of the intersection of availability and anticipability sets presented in Figure 6.12 together with the resulting primitive calls safely movable to the respective program point.

This example illustrates the previously presented equations. To be notice from the resulted availability and anticipability sets is the fact that $update(v, 2)$ is anticipable and available at the beginning of the application. This occurs since the horizon size is bigger than what the application initially exploited, i.e., the application through all its updates and releases only exploits the view sliding window up to a size of 1 event. As the horizon is defined with a size of 2, the initial code is suboptimal considering its frequent synchronization primitive calls and burst size of 1. Nevertheless, availability and anticipability analysis allow to predict $update(v, 2)$ as movable next to the existing $update(v, 1)$, allowing to reduce the number of synchronization calls and increasing burst size to the view horizon maximum of 2 record events, using the full view sliding window. The heuristics to detect and remove primitive redundant calls are executed after these PRE insertions.

PRE of synchronization primitives is mostly important to reduce the number of synchronizations per process iteration. Process blocking, as previously presented, expected such type of optimization considering its limited ability to perform Erbium primitives code motions. In the previously presented example, in Figure 6.8, blocking did not perform any code motion of the synchronization primitives, instead it copied the process code inside of a newly defined loop, iterating *BLOCKING_FACTOR* times. Such approach allowed blocking code transformations without primitives internal process analysis. As mentioned during the blocking optimization section, in order to take advantage of blocking it is necessary to reduce the number of synchronization calls, hoisting synchronization calls out of the process blocking created loop. This synchronization primitives hoisting is possible since the blocking optimization validates for the global inter-process validity (against deadlocks) and increases the involved views horizon sizes. With the increased view horizon size, PRE should be able to detect the synchronization redundancy and hoist the synchronization primitives out of the loop.

6.5 Erbium redundancies

Partial redundancy elimination pass starts by computing both availability and anticipability sets. By traversing the sets, the pass identifies possible code insertions introducing redundancies. The insertions also convert any partial redundant expression into fully redundant ones. Fully redundant statements are then removed.

But how are Erbium statements redundant to each other?

An Erbium primitive is redundant with respect to another if all the dependencies of the first are also dependent or "covered" by the second. Taking in consideration such notion, and for S_p being the set of all Erbium primitives in the process p, a statement redundancy can be verified through the following two equations:

$$a \text{ is redundant if } \forall c \in S_p : a \ \delta \ c \Rightarrow \exists b \in S_p : b \ \delta \ c \wedge b \in doms(a) \tag{6.10}$$

$$a \text{ is redundant if } \forall c \in S_p : a \ \delta \ c \Rightarrow \forall path \in paths(a, c), \exists e \in path : e \ \delta \ c \tag{6.11}$$

Although correct, this definition implies too many verifications, making it unrealistic for a compiler implementation. The equations verify for the two possible redundancy scenarios. Equation (6.10) finds a substituting primitive in a dominator of a. Dominator verification is already a simplification to finding a redundancy in all paths reaching a. Equation (6.11) refers to finding substituting primitives in all the possible paths from a to any of its dependents.

These equations are not intended to detect *occ* redundancies considering all the other non Erbium code data dependencies associated with the *occ* primitive. Its redundancy is detected by traditional PRE once the *occ* primitive gets converted into direct buffer memory accesses, as mentioned before.

Considering the precise Erbium synchronization semantics and what the dependencies really verify by, i.e., the relation of the primitives based on the interactions with the view sliding window, it should be possible to simplify these equations using PRE analysis (the availability and anticipability sets). Taking in consideration only the synchronization primitives (*update*, *release*, *stall* and *commit*) a primitive is redundant if there is other similar primitive call with a bigger or equal index in a protective place towards all dependents control-flow placement.

6.5.1 PRE redundancy detection

Availability and anticipability sets provide precise expectations for the type of primitives possibly insertable at all program points. An *update*, for example, generates (*gen*) the expectation of find or even to move a set of *occ* primitives in a forward execution flow. If an availability set includes a range of *occ* primitives, it is forcefully either generated from a *stall* or *update* primitive depending on the view type. This approach is usable to identify the same type of redundant primitives as in Equation 6.10. Considering how availability is computed, such optimization is more precise then the previous presented dominator simplification. Moreover, using the PRE availability set, it is possible to compute such type of redundancy, based on:

$$a \textit{ is redundant if } \max(avail\_gen[a]) \in avail\_in[a] \tag{6.12}$$

For the particular case of an *update* primitive, if its availability set at entry point (*avail_in*) contains the maximum index *occ* generated by this particular call, then it is obvious that either an equal or bigger index *update* is executed in a dominator or in all the possible paths to this program point. This verification guarantees the redundancy of the *update* call.

The Equation 6.11, although not defined using dominators, is symmetric to the redundancy detection in Equation 6.12. The existing PRE analysis already provide a symmetrical definition of availability, more precisely the anticipability.

The following equation formalizes this redundancy detection case using anticipability.

$$a \textit{ is redundant if } \max(antic\_gen[a]) \in antic\_out[a] \tag{6.13}$$

Figure 6.14 presents three distinct cases where anticipability is used for the release primitives redundancy detection. Rounded corner boxes are basic blocks. Dashed rectangles are the anticipability either at the beginning or end of the basic block, also known as *in* and *out* anticipability respectively, as defined in Equation 6.3. The ~~striked~~ primitives are the ones identified as redundant. The primitives responsible for the redundancy are marked with underline.

In the left most example, the *release* primitive in the uppermost basic block is identified as redundant considering that all its successor basic blocks contain higher index releases.

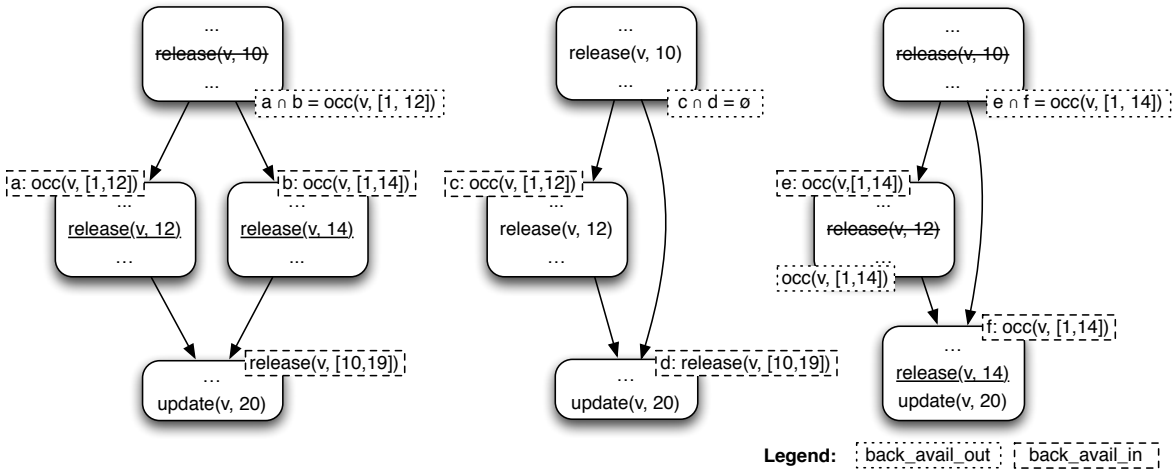


Figure 6.14: Redundancy detection for release primitives.

As anticipability is computed using an intersection of the successor sets of the basic blocks, only if all these successors contain a *release* in its reverse control-flow path, the respective program point would not contain a range of *occ*s in the set. Moreover, and as defined in the redundancy equation, the primitive is only redundant if its maximum *antic_gen* primitive index is in its out anticipability set. As the intersection of all the successors contain $occ(v, 10)$ (the maximum index for $antic\_gen[release(v, 10)]$), the primitive is marked as redundant.

In the next example, none of the *release* primitives is redundant. This is the case since there is one path for both the *release* primitives to one of its dependency sinks ($update(v, 20)$) without reaching another *release* primitive with a higher index. This approach detects this non redundant case flawlessly.

In the last example, as the bottom most basic block contains a *release* primitive before the *update*, both the releases in the remaining basic-blocks are identified as redundant.

6.5.2 Previous PRE example redundancy elimination follow-up

Figure 6.15 is the follow up of the availability and anticipability sets creation example in Figure 6.12. PRE is an iterative transformation where succeeding code transformations reveal further possible improvements to code. This particular example requires three iterations to conclude.

The first iteration is the insertion of the statement $s2$ right after the view connectivity. The availability and anticipability sets, validating for this first iteration, are presented in Figure 6.12.

The second iteration is the removal of the now obvious redundant *update* primitive calls (statements $s3$ and $s6$). The redundancy detection is based on Equation 6.12. As the inserted statement $s2$ adds to the availability set an *occ* in the range $[1, 2]$, and as $s3$ and $s6$ have no effect on the availability set, such primitives are classified as redundant. In other words, these primitive calls make no change to the view sliding window and for that matter are redundant.

The third and last iteration is the removal of the intermediate *release* primitive. The *release* primitive redundancy is not detected at the same iteration as the *update*s. This is

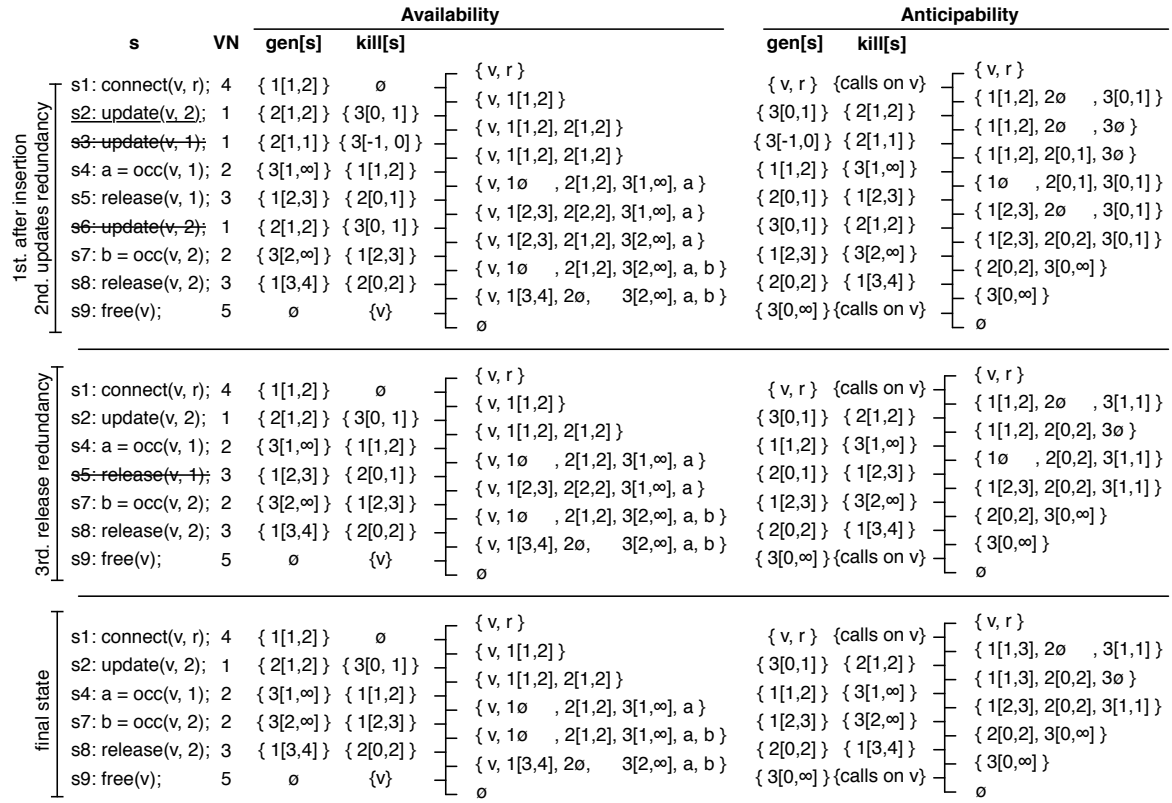


Figure 6.15: PRE redundancy detection example.

the case because of the impact the redundant *update* primitives have on the anticipability set. Once $s6$ is removed, the redundancy of $s5$ is easily detected through Equation 6.13.

6.5.3 Incomparable index synchronization reductions

Redundancy detection as presented until now is limited by the possibility to compare the index of the possible redundant primitives. However, there are particular scenarios where such comparison is not necessary, depending on the control-flow relation of the primitives.

More precisely, instead of removing one of the primitives, we can merge both by, at runtime, compute the maximum of both primitives index arguments. This approach is possible if there is no sliding window conflicting primitive call in the control-flow path between the two merged primitives. In the merge of two *update* primitives, we must guarantee that no affecting *release* primitive can execute between the two *update* calls. Moreover and considering inter-process dependencies, it must also verify that no dependent commit is present. Please notice that the merge is only necessary when the primitives are incomparable.

Merging primitives, although not detecting any redundancy, significantly reduces execution time by the overhead of one synchronization primitive.

6.5.4 Dead-code elimination

Dead code elimination (DCE) like redundancy detection can also be computed based on PRE analysis availability and anticipability sets. In the particular case of DCE, a primitive should

be considered dead code if it is not anticipable.

Let us consider $a = occ(v, i)$ primitive. If a is never used, it is detected through traditional DCE and removed, also removing the *occ* call statement. Before *occ* primitive is removed, this same primitive would anticipate that an *update* primitive would be available before this call. As the *occ* is no longer in the code, the *update* is no longer anticipable.

Existing primitives to which the primitive is neither available or anticipable are considered as dead code and can be removed.

6.6 Summary and next steps

In this chapter we shown how the Erbium intermediate representation can be used to optimize parallel streaming applications.

We first presented a study on the Erbium language component dependencies. This study permitted us to identify three different IR abstraction layers, each having its benefits, better matching the optimizations (code transformations) also presented in the chapter. The code conversions presented in Chapter 4 are responsible for the conversion between the different abstractions.

The inter-process and intra-process Erbium primitive relations are studied and represented as a collection of dependency equations. These equations define the interactions between the different primitives, used to extend partial redundancy detection and elimination to streaming data-flow programs. The effectiveness of our Erbium-level PRE depends on the availability of closed forms for scalar induction variables (scalar evolution analysis) for the primitives indexes. We only consider a few cases of index comparisons. But we believe that interesting redundancies, and other optimizations can be modeled by extending our approach beyond these few cases.

Taking advantage of the different abstraction layers, we also studied the adaptation of important task-level optimizations such as blocking and task fusion. These optimizations are further improved by PRE, reducing number of synchronizations and improving locality.

Although the Erbium IR was implemented in GCC, the more advanced optimizations studied in this chapter remain essentially theoretical, mostly because of the complexity of the full infrastructure, involving the PNG, interprocedural induction variables, and reengineering of a number of analyzes and optimizations in the compiler's middle-end.

Chapter 7

Conclusion

This thesis guides the reader through the key streaming parallelism components, more precisely a language definition, compilation approach and execution environment.

Chapter 1 introduces the thesis and states its main research problems, in considering the very large variety of languages, architectures and operating systems. It also shows how each of these computing abstractions increases the compilers and runtime libraries complexity, and how both of these are responsible for adapting parallel application written in high-level parallel languages to the target architectures and operating systems. The variety is the result of the many types different architecture and operating systems properties — for example, the architecture memory model with its consistency and coherency models or the operating system with its supporting threading interfaces and scheduling policies. In the chapter we conclude that streaming languages are among the best candidates to expose parallelism and data access patterns suitable for a big variety of architectures, having a much higher portability level considering its ability to capture applications (in)dependence and locality.

In Chapter 2, we present Erbium — the main "ingredient" in this thesis dissertation — a decoupled and expressive streaming language, also used as a compiler intermediate representation. The Erbium's language close to hardware semantics allows it to represent a large variety of high-level streaming languages, supporting decoupled communication and synchronizations, data peek and poke operations, broadcasting, work-splitting and process hand-overing. As a KPN extension, the language maintains its applicational determinism and modularity. Contrarily to KPN, it provides further modularity through its ability to perform data communication granularity adjustments resorting to process code loop transformations.

Supporting the Erbium language, Chapter 3 presents a low overhead x86 runtime library (libEr), designed with busy-waiting synchronizations. We explain the main concurrency issues arising with the implementation of Erbium runtime primitives, relating with the target architecture memory model. The presented implementation takes advantage of the x86 architecture total store ordering (TSO) memory consistency model, allowing the implementation of libEr not to rely on any memory barriers or atomic operations.

The same chapter also presents the libEr portability penalties associated with other weaker consistency memory model architectures — the introduction of memory barriers or atomic operations. We present the differences required by more lazy runtime implementations, most important for power efficiency as is the case in many general purpose embedded devices, such as smartphones or tablet devices. Busy-waiting is mainly useful for well bal-

anced applications and in target platforms where a subset of the system processors usage is saturated. This scenario relates to embedded system, targeted for specific applications — for example, digital signal processing (DSP) applications. Simultaneously we present the possibility to execute Erbium processes in user-level threads, explaining its scheduling benefits when comparing with kernel-level threads and the Linux scheduling policies.

The chapter also presents a set of precise experiments proofing the performance of our initial implementation best assumptions. Erbium's long running processes have very small overheads when comparing with the continuous task calling approach. We also compared the data communication granularity adjustments through synthetic benchmarks where we could identify that different architectures yield best performance at precisely tuned granularity parameters. Moreover, Erbium's merits are evaluated with the implementation of real world applications, exploiting all the available task and data-parallelism, and also out-performing other parallel languages, achieving 14.6× speedup in the GNU FMRadio application and 7.45× in a prototype 802.11a Wifi application from Nokia (results obtained in the AMD Opteron 16 core machine).

Chapter 4 presented a brief introduction to the software architecture of GCC, outlining its data structures and intermediate representations. It also shows how the Erbium intermediate representation is integrated in GCC, making use to GIMPLE original representation through the construction of the Erbium primitives as GCC builtins. Although builtins are about as verbose as the direct library calls introduced by any source-to-source compilation, the precise definition of these builtins and the pointer-specific attributes avoid the obfuscation of the Erbium semantics. For example, Erbium primitives can so be moved in respect to the non Erbium dependent code just like any surrounding pure function cans. Apart from Erbium's integration in a production compiler, the chapter also introduced an associated data structure, named Process Network Graph, exposing Erbium to the high-level semantical information available at the original language.

In Chapter 5 we present the lowering of a Streaming extension of OpenMP into Erbium. The high-level semantics of the language allows the lowering phase to collect static information enabling optimizations such as blocking, process fusion, record fusion and static scheduling, as presented in Chapter 6. These rich language properties are provided to compilers through the Process Network Graph.

Streaming OpenMP lowering involves converting short-living tasks into persistent processes, a redesign of process instantiation (converting explicit task instantiation into process execution based on data availability), the decoupling of synchronization and communication, and the conversion of stream/window variables into Erbium's event accesses (*occ* operations).

Chapter 6 formalizes the dependencies between synchronization and data communication primitives, also addressing inter-process synchronization and communication dependencies. The study of inter-process dependencies allows us to identify three distinct abstraction levels associated with different families of optimizations — inter-process analysis optimizations, synchronization-related optimizations, and the traditional optimizations for the sequential code and Erbium data communication/access primitives. The chapter presents examples of optimizations/transformations implementable using Erbium IR and considering that sufficient Process Network Graph information is available.

The chapter also presents an extension to the partial redundancy elimination (PRE) pass in GCC, enabling Erbium primitives code motion and redundancy optimizations. This extension is possible thanks to the dependencies study, clean set of primitives and the Erbium's

very close to hardware semantics. Through the extension of availability and anticipability sets, as presented in the chapter, we prove that Erbium's code motion and redundancy detection is merely a specialization of mainstream PRE optimizations. In any case, PRE is limited by the possibility to compare the Erbium primitives monotone index arguments using induction variable evolution prediction.

7.1 Future work

The presented ideas and implementations are not yet at a stable level and much must be done before such improvements can be merged into a mainstream compiler distribution. Moreover, many of the less detailed topics in the document, such as PNG and example optimizations require much deeper studies.

Many of the presented optimizations tightly depend on the availability of specific PNG collected information. This is not an Erbium limitation but rather a consequence of its high expressiveness and low-level semantics, where optimization improvements such as the presented partial redundancy elimination are possible.

PNG data structure must be tightly related with the Erbium intermediate representation. Transformations to the intermediate representation weaken this relation to the point where PNG can become meaningless for existing Erbium code. For this reason, it is necessary to also perform future studies of the impact Erbium IR transformations have to the PNG data structure.

The presented partial redundancy elimination improvements depend on the comparison of the Erbium primitive index arguments. The presented solution is limited to the availability of scalar evolution for the particular variables. Scalar evolution for this purpose works only with simple predictable loop iterations, supporting only simple Erbium code motions. Our belief is that there exist other less restrictive ways to perform this verifications not relying on induction variable analysis.

By contributing to languages, runtimes, compilation and optimizations, this thesis left many open research axis.

On the language perspective it would be interesting to continue the validation study for the language support for more dynamic load balancing scenarios. For example, to distribute work between similar consumer worker processes solely by there availability.

The runtime subject requires further work in the validation of the presented distributed memory design and extra experiments for different thread systems and scheduling policies.

The compilation/optimizations subject has open for research topics, such as: exploring new possible parallel optimizations and heuristics; the partial redundancy elimination extension requirement in a less intrusive manner, predicting availability or anticipability set for Erbium primitives — based on more efficient ways to compare index arguments; and a PNG data structure definition and stronger correlation to the Erbium intermediate representation.

7.2 Perspectives

The gap between languages and architectures computer abstractions is enormous, each having there distinctive and independent set of problems. This is the reason why most research in this topics happens independently, targeting only a specific type of language or architecture. This is one of the reasons for the long list of source-to-source compiled high-level parallel languages

and the long list of designed architectures that never had a reliable compiler support, being only used for research purposes.

This differentiation of both these research fields exists thanks to the complexity of the intermediate layers, more precisely the complexity of mainstream compilers, making most architecture and language designers to escape complex compiler implementations. In our perspective, compilers can no longer be threated as the "black sheep" of these research fields and an extra effort must be done to recover the pace against the latest years improvements from architectures and parallel languages.

Compiler development is a very frustrating and challenging job. Its development occurs in very small incremental stages, guaranteeing newly transformations do not impact code correctness and performance for any variety of applications. This thesis presented a new language, runtime library, compiler modifications and optimization improvements, providing a portability layer to streaming data-flow languages. The defined Erbium components (language, intermediate representation and runtime library) are the missing abstractions filling the gap between parallel streaming languages and target architectures. The compiler job is to transform and optimize the code within these abstractions.

The presented evolutions in this thesis are a first step towards the unification of mainstream compilers, streaming parallel languages and its optimizations. However, not all of the presented work has an actual implementation or is easily achievable.

We believe that future compilers not only should try to detect code parallelism but also support explicit parallelism, targeting code generation to specific and well tunned target architecture specific runtime library implementations, while still being able to use high-level language properties to perform optimizations, as well as more general parallel and sequential optimizations. For that matter compilers should be capable to express (represent) the explicit parallelism from a vast number of high-level languages.

With the introduction of Erbium and its distinctive components, we not only can improve the compilation environment for streaming data-flow languages but also significantly reduce the existing distance between these programming languages and hardware, possibly leading to a more portable support for streaming data-flow languages.

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] ACOTES-Advanced Compiler Technologies for Embedded Streaming . URL `http://www.hitech-projects.com/euprojects/ACOTES/`.

[2] The LLVM Compiler Infrastructure. URL `http://llvm.org/`. 11

[3] The Message Passing Interface (MPI) standard. URL `http://www.mcs.anl.gov/research/projects/mpi/`. 88

[4] OpenCL - The open standard for parallel programming of heterogeneous systems. URL `http://www.khronos.org/opencl/`. 13

[5] The StreamIt Language. URL `http://groups.csail.mit.edu/cag/streamit/`.

[6] *OpenMP Architecture Review Board*. OpenMP Application Program Interface - Version 3.1. July 2011. 106

[7] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison- Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. ISBN 0-201-10194-7. 98, 99, 100, 113

[8] Ghiath Al-Kadi and Andrei Terechko. A hardware task scheduler for embedded video processing. In *Proceedings of the 4th International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC'09)*, Paphos, Cyprus, 2009. 19

[9] Marco Aldinucci, Massimiliano Meneghin, and Massimo Torquati. Efficient Smith-Waterman on multi-core with FastFlow. In *Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP'10)*, pages 195–199, Pisa, 2010. 52

[10] Andrew W. Appel. *Modern Compiler Implementation in C*. Cambridge University Press, New York, NY, USA, 1998. ISBN 0-521-58390-X. 148, 149

[11] Arvind, Rishiyur S. Nikhil, and Keshav Pingali. I-Structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11 (4):598–632, October 1989. doi: 10.1145/69558.69562. 18, 19

[12] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'11)*, pages 487–498. ACM Request Permissions, January 2011. doi: 10.1145/1926385.1926442.

[13] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Maik Nijhuis. Exploiting the Cell/BE architecture with the StarPU unified runtime system. In *Proceedings of the 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS'09)*, pages 329–339, 2009. doi: 10.1007/978-3-642-03138-0{_}36.

[14] Arnaldo Azevedo, Cor Meenderinck, Ben Juurlink, Andrei Terechko, Jan Hoogerbrugge, Mauricio Alvarez, Alex Ramírez, and Mateo Valero. Parallel H.264 decoding on an embedded multicore processor. In *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers (HiPEAC'09)*, Paphos, Cyprus, January 2009. doi: 10.1007/978-3-540-92990-1{_}29. 41, 80

[15] Woongki Baek, Chi Cao Minh, Martin Trautmann, Christos Kozyrakis, and Kunle Olukotun. The OpenTM Transactional Application Programming Interface. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT'07)*, pages 376–387, 2007. doi: 10.1109/PACT.2007.74. 106, 108

[16] Greet Bilsen, Marc Engels, Rudy Lauwereins, and J. A. Peperstraete. Cyclo-static data flow. In *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP'95)*, pages 3255–3258, Detroit, Michigan, 1995. 10

[17] OpenMP Architecture Review Board. OpenMP Application Program Interface, July 2011. URL `http://www.openmp.org/mp-documents/OpenMP3.1.pdf`. 12, 95, 106, 117

[18] Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation (PLDI'08)*, pages 68–78, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: 10.1145/1375581.1375591. 96

[19] Paul Carpenter, David Ródenas, Xavier Martorell, Alex Ramírez, and Eduard Ayguadé. A streaming machine description and programming model. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS'07)*, pages 107–116, Samos, Greece, July 2007. 117

[20] Paul Caspi and Marc Pouzet. Synchronous Kahn Networks. In *Proceedings of the first ACM SIGPLAN international conference on Functional programming (ICFP'96)*, pages 226–238, 1996. ISBN 0-89791-770-7. doi: 10.1145/232627.232651. 19

[21] Albert Cohen, Louis Mandel, Florence Plateau, and Marc Pouzet. Abstraction of clocks in synchronous data-flow systems. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems (APLAS' 08)*, Bangalore, India, December 2008.

[22] INMOS Corp. *Occam Programming Manual*. Prentice Hall, 1984. ISBN 0136292968. 19

[23] David E. Culler and Arvind. Resource requirements of dataflow programs. In *Proceedings of the 15th Annual International Symposium on Computer architecture (ISCA'88)*, pages 141–150, May 1988.

[24] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*

(POPL'89), pages 25–35, New York, NY, USA, 1989. ACM. ISBN 0-89791-294-2. doi: 10.1145/75277.75280. 100

[25] Jack B. Dennis and Guang R. Gao. An efficient pipelined dataflow processor architecture. In *Supercomputing (SC'88)*, pages 368–373, 1988.

[26] Ulrich Drepper. What every programmer should know about memory, November 2007. URL `http://www.akkadia.org/drepper/cpumemory.pdf`. 75

[27] Ulrich Drepper. Futexes are Tricky. Technical report, 2009. 72

[28] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing (PODC'04)*, pages 50–59, New York, NY, USA, 2004. ACM. ISBN 1-58113-802-4. doi: 10.1145/1011767.1011776. 64

[29] Cédric Fournet and Georges Gonthier. The Reflexive Chemical Abstract Machine and the Join-Calculus. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '96)*, pages 372–385, St. Petersburg Beach, Florida, January 1996. ACM. doi: 10.1145/237721.237805. 19

[30] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI'98)*, pages 212–223, Montreal, Quebec, June 1998. 84, 86

[31] FSF. GNU Compiler Collection (GCC) Internals Manual, 2010.

[32] FSF. Gnu compiler collection (GCC), 2011. 11

[33] John Giacomoni, Tipp Moseley, and Manish Vachharajani. FastForward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In *Proceedings of the The 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08)*, pages 43–52, Salt Lake City, Utah, 2008.

[34] Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems (ASPLOS-XII)*, pages 151–162, San Jose, California, 2006. doi: 10.1145/1168857.1168877. 38, 79, 95

[35] Rajiv Gupta and Sunah Lee. Exploiting Parallelism on a Fine-Grain MIMD Architecture Based Upon Channel Queues. *International Journal of Parallel Programming*, 21(3): 169–192, June 1992. doi: 10.1007/BF01408554. 19

[36] Wolfgang Haid, Lars Schor, Kai Huang, Iuliana Bacivarov, and Lothar Thiele. Efficient execution of Kahn process networks on multi-processor systems using protothreads and windowed FIFOs. In *Workshop on Embedded Systems for Real-Time Multimedia (ESTImedia'09)*, pages 35–44, Grenoble, France, October 2009. 19

[37] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991. 19

[38] Robert H. Halstead Jr. MULTILISP: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(4):501–538, October 1985. doi: http://doi.acm.org/10.1145/4472.4478. 18, 20

[39] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: a million lines of C code in a second. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation (PLDI '01)*, pages 254–263, New York, NY, USA, 2001. ACM. ISBN 1-58113-414-2. doi: 10.1145/378795.378855. 102

[40] Tomas Henriksson and Pieter van der Wolf. TTL Hardware Interface: A High-Level Interface for Streaming Multiprocessor Architectures. In *Workshop on Embedded Systems for Real-Time Multimedia (ESTImedia'06)*, pages 107–112, Seoul, Korea, October 2006. 87

[41] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985. ISBN 0-13-153271-5. 19

[42] H. Peter Hofstee. Power Efficient Processor Architecture and The Cell Processor. In *11th International Symposium on High-Performance Computer Architecture*, pages 258–262. IEEE, 2005. ISBN 0-7695-2275-0. doi: 10.1109/HPCA.2005.26. 2, 6, 87

[43] *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, Santa Clara, CA, USA, 2006. 58

[44] Gilles Kahn. The semantics of a simple language for parallel programming. In J L Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, August 1974. North Holland, Amsterdam. 10, 17

[45] Robert Kennedy, Sun Chan, Shin-Ming Liu, Raymond Lo, Peng Tu, and Fred Chow. Partial redundancy elimination in SSA form. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(3):627–676, May 1999. doi: 10.1145/319301.319348. 149

[46] Chinhyun Kim, Jean-Luc Gaudiot, and Wlodek Proskurowski. Parallel Computing with the Sisal Applicative Language: Programmability and Performance Issues. *Software, Practice and Experience*, 26(9):1025–1051, September 1996. doi: 10.1002/(SICI)1097-024X(199609)26:9⟨1025::AID-SPE48⟩3.0.CO;2-H. 18, 19

[47] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Lazy code motion. *ACM SIGPLAN Notices*, 39(4):460–472, 2004. doi: 10.1145/989393.989439. 149

[48] Costas Kyriacou, Paraskevas Evripidou, and Pedro Trancoso. Data-Driven Multithreading Using Conventional Microprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 17(10):1176–1188, 2006. 19, 87

[49] Fabrice Le Fessant and Luc Maranget. Compiling Join-Patterns. *Electronic Notes in Theoretical Computer Science*, 16(3), 1998. 19

[50] Edward Ashford Lee and David G. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Transactions on Computers*, 36(1):24–35, January 1987. doi: 10.1109/TC.1987.5009446. 10

[51] Edward Ashford Lee and Alberto Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, 1998. 17

[52] Vladimir Marjanovic, Jesús Labarta, Eduard Ayguadé, and Mateo Valero. Effective communication and computation overlap with hybrid MPI/SMPSs. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'10)*, 2010. doi: 10.1145/1693453.1693502. 87, 117

[53] Paul E. McKenney. Memory ordering in modern microprocessors, Part I. *Linux Journal*, 2005(136), August 2005. 5, 6

[54] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures (SPAA'02)*, pages 73–82, 2002. doi: 10.1145/564870.564881. 64

[55] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, I and II. *Information and Computation*, 100(1):1–40 and 41–77, 1992. 19

[56] Cupertino Miranda, Philippe Dumont, Albert Cohen, Marc Duranton, and Antoniu Pop. ERBIUM: a deterministic, concurrent intermediate representation for portable and scalable performance. In *Proceedings of the 7th ACM international conference on Computing frontiers (CF'10)*, pages 119–120. ACM, May 2010. doi: 10.1145/1787275.1787312.

[57] Cupertino Miranda, Antoniu Pop, Philippe Dumont, Albert Cohen, and Marc Duranton. Erbium: a deterministic, concurrent intermediate representation to map data-flow tasks to scalable, persistent streaming processes. In *Proceedings of the 2010 international conference on Compilers, architectures and synthesis for embedded systems (CASES'10)*, pages 11–20, New York, NY, October 2010. ACM Request Permissions. doi: 10.1145/1878921.1878924.

[58] Harm Munk, Eduard Ayguadé, Cédric Bastoul, Paul Carpenter, Zbigniew Chamski, Albert Cohen, Marco Cornero, Philippe Dumont, Marc Duranton, Mohammed Fellahi, Roger Ferrer, Razya Ladensky, Menno Lindwer, Xavier Martorell, Cupertino Miranda, Dorit Nuzman, Andrea Ornstein, Antoniu Pop, Sebastian Pop, Louis-Noël Pouchet, Alex Ramírez, David Ródenas, Erven Rohou, Ira Rosen, Uzi Shvadron, Konrad Trifunovic, and Ayal Zaks. ACOTES Project: Advanced Compiler Technologies for Embedded Streaming. *International Journal of Parallel Programming*, pages 1–54, April 2010. 80

[59] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: Efficient Deterministic Multithreading in Software. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems (ASPLOS '09)*, pages 97–108, Washington, DC, March 2009. doi: 10.1145/1508244.1508256.

[60] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. Automatic Thread Extraction with Decoupled Software Pipelining. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture (MICRO'38)*, pages 105–118, 2005. 19, 85

[61] Scott Owens, Susmit Sarkar, and Peter Sewell. A Better x86 Memory Model: x86-TSO. In *Theorem Proving in Higher Order Logics*, pages 391–407, Springer Berlin/ Heidelberg, 2009. 58

[62] David J. Pearce, Paul H. J. Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis of C. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(1), November 2007. 102

[63] Josep M. Pérez, Pieter Bellens, Rosa M. Badia, and Jesús Labarta. CellSs: Making it easier to program the Cell Broadband Engine processor. *IBM Journal of Research and Development*, 51(5):593–604, 2007.

[64] Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. Hierarchical Task-Based Programming With StarSs. *International Journal of High Performance Computing Applications*, 23(3):284–299, August 2009. doi: 10.1177/1094342009106195. 117

[65] Antoniu Pop and Albert Cohen. Preserving high-level semantics of parallel programming annotations through the compilation flow of optimizing compilers. In *Proceedings of the 15th Workshop on Compilers for Parallel Computers (CPC'10)*, Vienna, Autriche, 2010. Centre de recherche en informatique - CRI , ALCHEMY - INRIA Saclay - Ile de France. 108

[66] Antoniu Pop and Albert Cohen. A Stream-Comptuting Extension to OpenMP. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC'11)*, pages 5–14, New York, NY, January 2011. 117, 120, 121

[67] Antoniu Pop, Sebastian Pop, Harsha Jagasia, Jan Sjödin, and Paul H. J. Kelly. Improving GNU Compiler Collection Infrastructure for Streamization. In *Proceedings of the 2008 GCC Developers' Summit*, pages 77–86, June 2008. 38

[68] Antoniu Pop, Sebastian Pop, and Jan Sjödin. Automatic Streamization in GCC. In *Proceedings of the 2009 GCC Developer's Summit*, Montreal, Quebec, June 2009. 117

[69] Sebastian Pop. Analysis of induction variables using chains of recurrences: extensions. Master's thesis, Université Louis Pasteur, Strasbourg, July 2003. 154

[70] Sebastian Pop, Philippe Clauss, Albert Cohen, Vincent Loechner, and Georges-André Silber. Fast recognition of scalar evolutions on three-address SSA code. Technical report, 2004. 100, 154

[71] Sebastian Pop, Albert Cohen, and Georges-André Silber. Induction Variable Analysis with Delayed Abstractions. In Tom Conte, Nacho Navarro, Wen-mei Hwu, Mateo Valero, and Theo Ungerer, editors, *High Performance Embedded Architectures and Compilers*, pages 218–232. Springer Berlin / Heidelberg, 2005. ISBN 978-3-540-30317-6. doi: 10.1007/11587514{_}15. 100

[72] Sebastian Pop, Albert Cohen, Cédric Bastoul, Sylvain Girbal, Georges-André Silber, and Nicolas Vasilache. GRAPHITE: Loop optimizations based on the polyhedral model for GCC. In *Proceedings of the 4th GCC Developper's Summit*, pages 179–198, Ottawa, Canada, June 2006. 104

[73] Martin C. Rinard and Monica S. Lam. The design, implementation and evaluation of Jade. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(3): 483–545, May 1998. doi: http://doi.acm.org/10.1145/291889.291893. 19

[74] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. The semantics of x86-cc multiprocessor machine code. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, pages 379–391, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-379-2. doi: 10.1145/1480881.1480929. 58

[75] Magnus Själander, Andrei Terechko, and Marc Duranton. A Look-Ahead Task Management Unit for Embedded Multi-Core Architectures. In *Proceedings of the 2008 11th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools (DSD'08)*, Parma, Italy, September 2008. 19

[76] Kyriakos Stavrou, Marios Nikolaides, Demos Pavlou, Samer Arandi, Paraskevas Evripidou, and Pedro Trancoso. TFlux: A Portable Platform for Data-Driven Multithreading on Commodity Multicore Systems. In *Proceedings of the 2008 37th International Conference on Parallel Processingg (ICPP'08)*, pages 25–34, Portland, Oregon, September 2008. doi: 10.1109/ICPP.2008.74. 117

[77] Robert Stephens. A Survey of Stream Processing. *Acta Inf.*, 34(7):491–541, 1997. 9

[78] Sander Stuijk. Concurrency in Computational Networks. Master's thesis, Technische Universiteit Eindhoven (TU/e), 2002. 80

[79] Andrew S. Tanenbaum. *Modern operating systems.* Prentice Hall, 2nd edition, 2001. ISBN 0-13-031358-4. 8

[80] William Thies and Saman Amarasinghe. An Empirical Characterization of Stream Programs and its Implications for Language and Compiler Design. In *International Conference on Parallel Architectures and Compilation Techniques (PACT'10)*, Vienna, Austria, September 2010. 80

[81] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proceedings of the 11th International Conference on Compiler Construction (CC'02)*, pages 179–196, April 2002. 19, 38

[82] John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing (PODC'95)*, pages 214–222, New York, NY, USA, 1995. ACM. ISBN 0-89791-710-3. doi: 10.1145/224964.224988. 64

[83] Thomas Vandrunen. *Partial Redundancy Elimination for Global Value Numbering.* PhD thesis, Purdue University, August 2004. 149

[84] Thomas Vandrunen and Antony L. Hosking. Value-based partial redundancy elimination. In *Compiler Construction, 13th International Conference (CC 2004)*, pages 167–184, Barcelona, Spain, 2004. 149

[85] Ian Watson and John R. Gurd. A Practical Data Flow Computer. *IEEE Computer*, 15 (2):51–57, 1982.