Notifying Memories for Dataflow Applications on Shared-Memory Parallel Computer

Alemeh Ghasemi

DOCTORAT BRETAGNE LOIRE / MATHSTIC

Université Bretagne Sud

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE BRETAGNE SUD

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par

« Alemeh GHASEMI »

« Notifying Memories for Dataflow Applications on Shared-Memory Parallel Computer »

Thèse présentée et soutenue à « Lorient », le « 18 mai 2022 »
Unité de recherche : Lab-STICC
Thèse N° : « 620 »

Rapporteurs avant soutenance :

Jocelyn SÉROT	Professeur - Université Clermont Auvergne
Frédéric ROUSSEAU	Professeur - Université Grenoble Alpes

Composition du Jury :

Président :	Jean-François NEZAN	Professeur - INSA Rennes
Examinateurs :	Jocelyn SÉROT	Professeur - Université Clermont Auvergne
	Frédéric ROUSSEAU	Professeur - Université Grenoble Alpes
	Alix MUNIER KORDON	Professeur - Sorbonne Université
	Midia RESHADI	Research Fellow - Trinity College Dublin
Dir. de thèse :	Jean-Philippe DIGUET	DR CNRS - IRL Crossing
Encadrant :	Kevin MARTIN	Maître de conférences - Université de Bretagne-Sud

Acknowledgement

I would like to thank my supervisors, Dr. Kevin Martin and Dr. Jean-Philippe Diguet, for their patience, guidance, and support. I have significantly benefited from their knowledge and meticulous editing. I am incredibly grateful that they have conducted this thesis over the years.

I am also grateful for the help of the committee members, Professor Jean-François Nezan and Dr. Midia Reshadi. Their encouraging words and thoughtful, detailed feedback have been significant to me. Additionally, I greatly appreciate the reviewers of my thesis, who so generously took time out of their schedules to review my research and make this project possible.

Lastly, I thank my family for their endless support. They have always stood behind me, and this was no exception. I could not have done it without them.

Table of Contents

List of Figures

List of Tables

INTRODUCTION

The performances of uni-core processors have been constantly improving for decades owing to Instruction-Level Parallelism (ILP) and rising hardware clock frequencies. However, the performance of large single-core processors no longer scales since only a limited amount of parallelism can be achieved by employing conventional superscalar instruction techniques in a typical instruction stream [32, 85]. Moreover, elevating the clock speed on today's processors is limited by the power dissipation, which indicates the failure of Dennard scaling [25]. This power dissipation becomes prohibitive in all, but water-cooled systems [104]. Considering these problems and the large number of transistors available on today's microprocessor chip [99], a large and complex uni-core processor was too costly to design and debug, so this design trend was unsustainable. However, higher performance is still demanded, as manifested by predictions from the ITRS Roadmap 2011 [66], forecasting a need for 300x more performance by 2022. To avoid complex, unmanageable, and power-hungry designs while pushing for more performance, the chip designers have shifted to a new processor design paradigm: multi-core processors. Multi-core processors are high-performance computer systems built up by embedding several processing units as cores in the same processor, which communicate through an interconnection NoC (Network on Chip) or bus [44, 104].

The main advantage of multi-core systems is that they provide high-performance, adding more parallel resources (cores) rather than increasing clock frequency while maintaining the power characteristics [24]. Multi-core processors provide task-level rather than instruction-level parallelism. It means the application workload can be divided into parallel tasks that can run concurrently on the different cores while providing a faster execution of the application. As the complexity and variety of the applications have increased, the requirement for more performance and general-purpose programmability has grown. Hence, general-purpose multi-core processors have broadly been adopted in all industry segments, including digital signal processing and embedded systems, and are no longer restricted to the High Performance Computing (HPC) industry. Unfortunately, the performances of the applications do not scale automatically as newer processors providing more parallel cores are released. Thus, to make the most of multi-core processors' architectures, pro-

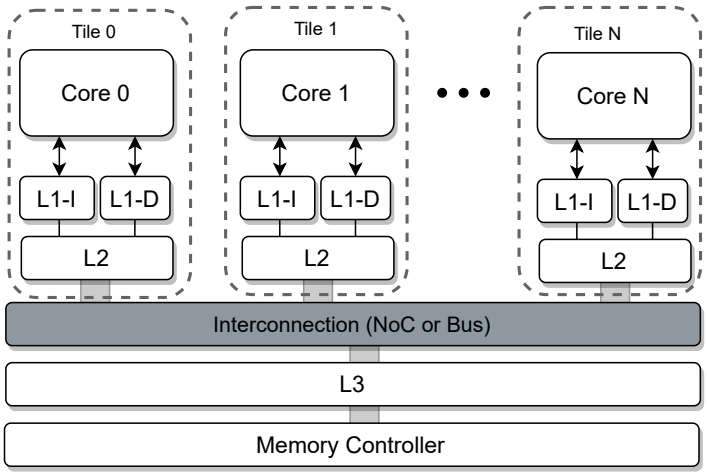


Figure 1 – Example of the architecture of a typical SMP. In this example, each core has a private L1 cache for instruction and a private L1 cache for data, a private L2 cache for both instructions and data. All the cores share the same L3 cache. The memory controller interfaces with the off-chip main memory.

gramming methods should evolve from a sequential to a parallel design approach. In a parallel program, the programmer, by dividing the application's code into tasks, expresses task-level parallelism (TLP), which can explicitly exploit the parallel hardware processing units of multi-core processors executing concurrently [105].

Symmetric Shared-memory Multiprocessor (SMP)

Symmetric Shared-memory Multiprocessor (SMP) is an extensively used high-performance multi-core processor architecture as it helps the programmer to simplify the transformation from sequential to parallel programs, by preserving a single image of memory shared across the entire parallel cores. Using a global virtual address space minimizes the changes from a single processor machine to an SMP [131]. In this context, an application is composed of multiple threads that can run in parallel, TLP standing conveniently also for Thread-Level Parallelism.

A typical SMP includes several cores/processors all interconnected with a common shared main memory via an interconnection network (e.g., NoC or a bus). Figure 1 shows the architecture of a typical SMP. Due to technological limitations, the main memory and the cores are not in the same chip. Ideally, the two chips follow the same pace. Unfortunately, the disparity between the speed of processor cores and off-chip memory can be

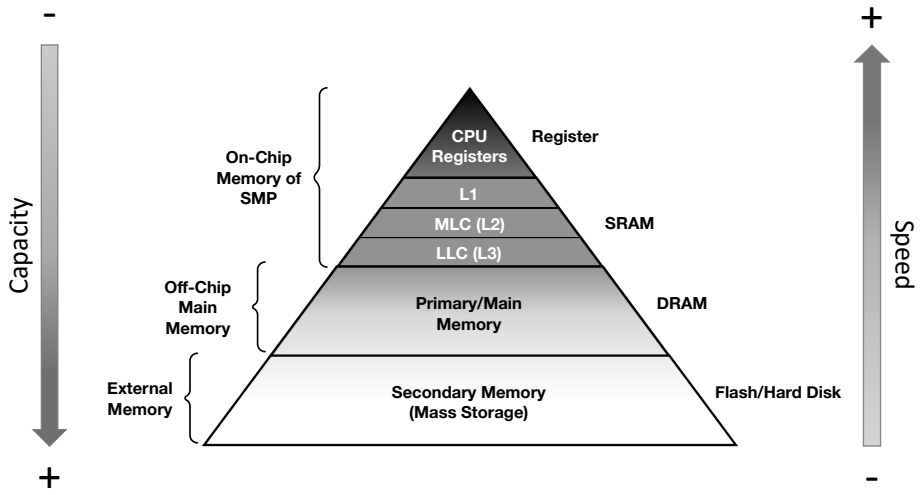


Figure 2 – Pyramid of memory hierarchy.

in the order of hundreds of cycles. The performances are limited by the memory latency. This is called the memory wall [94, 121, 145]. To circumvent this limitation, microprocessors, in general, and SMPs, in particular, use hardware-managed on-chip storage, called caches, to keep data within the chip to avoid such long accesses to the off-chip memory. The larger the cache, the lower the speed. Thus, typically, caches have been implemented in a hierarchical manner. Smaller and faster caches are close to the microprocessor while they are backed by two or three levels of larger caches. Frequently requested data is stored and accessed from a high-bandwidth and low latency cache which is usually a private local Level-1 (L1) cache of each processor core in the SMP. The lower levels of the caches (L2 and L3) are shared between the cores. Figure 1 shows an organisation where the L2 cache is private to each core, and the L3 cache is shared. The shared caches are slower both due to the size and the extra interconnection that is necessary to connect them to multiple cores, which also limits the bandwidth for sharing the data. This hierarchy of hardware-managed caches that are completely transparent to the developer is the key enabler of the illusion to access to a large and fast off-chip main memory.

The full memory hierarchy, from the registers of the processor to the hard-disk drive or the cloud, is usually represented in the form of a pyramid, with high speed but low capacity (and expensive) memory at the top, and low speed with high capacity (and cheap) memory at the bottom. Figure 2 presents the pyramid of memory hierarchy.

Caches exploit two forms of locality in the data accesses: temporal and spatial. Temporal locality means that when a memory location is accessed, it will be accessed in the

near future again. Spatial locality refers to the property that when a location is accessed, the addresses in its vicinity will also be accessed in the near future. Exploiting these properties, the caches bring a block (e.g., 32 consecutive words) from memory and store it when a single location is accessed. Because of the temporal locality of the accesses, when the same location is accessed again, it is already present in the caches (called a hit), and the long latency access to the off-chip memory is not necessary. Since a block is fetched and stored, accesses to the neighboring locations will also be hit in the cache avoiding access to the main memory. By exploiting spatial and temporal locality, caches reduce the effective latency of loads and stores. But the threads running on the cores of the SMP also can change concurrently the values in the caches instead of waiting for the modification to get published to the main memory. This poses a challenge when the data is shared between the threads running on different cores. A thread that is reading a value from its local cache may not see the writes done to the same location by another thread that modified the value of the same address in its own local cache. To keep the values coherent across these threads, a cache coherence protocol is needed.

A cache-coherent parallel processor (e.g., an SMP) must provide the following guarantees. Changes to the shared data, even when cached, shall become visible to all the threads on all the cores within a bounded (and typically short period of) time. This guarantee is achieved through a cache coherence protocol that, in almost all modern machines, is implemented using an invalidation-based mechanism. Such protocols, which are typically implemented in the cache controllers, maintains the invariant that there is at most one writable copy of any given cache block in the system. When the writable copy exists in one of the caches, there are no other valid copies in the other caches. Cache coherency and its algorithms has and are still being studied extensively in the literature [129]. Most protocols inherit from the original four-state protocol devised by Goodman [53]: each cache block is either (1) invalid, (2) shared (read-only), (3) exclusive (written exactly once, and up-to-date in memory), or (4) modified (written more than once, and needs to update the memory). To maintain the aforementioned at-most-one-writable-copy invariant, the coherence protocol needs to invalidate (evict) any shared block in other cores' caches before updating it with a new value. When the caches are connected through a bus, invalidation is straightforward and is performed through a broadcast message. When the caches are connected through a network-on-chip or point-to-point connection, the coherence protocol typically maintains a form of directory that locates all other copies of a shared block. The invalidation principle causes the accesses to the data to miss in the cache due to the

coherence protocol. Coherence protocol can cause loads and stores to miss in the cache since another core invalidates a previously cached block. Without code restructuring and a proper mapping of the threads on the cores, coherence misses are inevitable when some threads on a core writes and others on other cores read the same cache block in a small time window. To achieve high-performance execution for parallel programs, the threads need to exhibit thread locality in addition to temporal and spatial locality. Thread locality is referred to the property that modification to data in one thread is isolated from another thread at a given time. Thread locality reduces the aforementioned coherence cache misses that are due to shared data modifications and dependencies. Coherence can be managed by software or hardware mechanisms. In this thesis, we consider hardware-based coherence mechanisms.

Besides coherence and its considerations, caches need also to guarantee that accesses to some location will be visible in some form of consistent order to all the threads. Unlike coherence, the memory consistency model is exposed to the software and needs to be considered during execution. The memory consistency model is the degree to which accesses to the memory can or cannot be assumed to be published in a particular order. On a single-core machine, it is relatively straightforward to ensure that instructions, in general, and loads/stores, in particular, appear to complete in the program order. Ideally, it is desirable to ensure a similar consistent order on a parallel machine (e.g., an SMP) such that memory accesses, system-wide, would appear to constitute an interleaving of the completion order of the accesses on different cores. However, this sort of sequential consistency imposes significant performance overheads [77]. Therefore, the majority of the real-world systems implement a relatively more relaxed memory consistency model that can potentially exhibit inconsistency. That is to say, memory accesses by different threads, or to different locations by the same thread, may seem "out of order" from the perspective of threads that are running on the other cores. To this end, it is to the programmer or the compiler to ensure a specific order by means of synchronization. This is accomplished by using a set of special instructions that are more strongly ordered compared to other "ordinary" instructions as they force the core to temporally stop the execution and wait until the previous instructions are completed. These instructions are an essential part of synchronization algorithms on any non-sequentially consistent machine.

Execution on an SMP requires breaking up the code into multiple parallel threads that compute on shared data in the memory. Sharing data is how the threads communicate and satisfy their data dependencies. A producer thread writes to a shared location in

the memory where the dependent consumer thread can read the necessary data from. Hence, the order of writes and reads is a crucial constraint for the correct computation of data of a multi-threaded programs on SMPs. A reader may access to the (old and wrong) value before the writer effectively commits the new value, this is called a race condition. As such, threads need to *synchronize* before accessing the shared data to avoid race conditions that lead to incorrect results. Such a data that needs to be protected is called a critical resource. Synchronization techniques usually require serializing the execution of parallel threads to impose the necessary order. From an abstract and fundamental point of view, only one thread is allowed to write or read to a critical resource. From the software's perspective, the region of the thread code that accesses the critical resource is called critical section. The memory accesses to these critical sections need to be synchronized to ensure the correct data exchange between dependent threads. At a high level, a thread is said to *lock* the critical section and exclusively enters it to finish its writes or reads before any other thread. At the hardware level [108], Atomic Memory Operation (AMO) is the ultimate operation for the software-based lock algorithms that enables the threads to mutually exclusively enter the critical section and consistently share data. Software semaphore is an example of software-based synchronization primitive to control access of multiple threads to the shared critical data. In semaphore synchronization technique, threads utilize two functions to modify the semaphore value `wait()` and `signal()`, but the functions allow only one thread to change the value at a specific time, i.e., two threads cannot simultaneously change the semaphore value. There are two types of semaphores; counting semaphores and binary semaphores.

The mutually exclusive execution inherently leads to the serialization of threads which in turn looses all the benefits of parallel execution. In addition, the necessity and prevalence of caches in the SMPs exacerbates the overheads since locking and synchronization highly involve the coherence protocol in the cache subsystem. Indeed, preserving consistency and coherency of the lock values is mandatory and requires cache coherency protocol messages [128, 141]. For instance, when a thread running on one of the cores in the SMP, is accessing the shared data, the cache coherency protocol needs to invalidate the data in the caches of the other cores. This is just an instance of the significant number of coherency messages that need to be exchanged between the caches to allow the threads to enter the critical section one by one. This coherency procedure (a.k.a., True sharing [136]) imposes performance overhead that could cost more than one hundred cycles [141]. As the number of cores/communication pairs increases on a chip, cache line ping-pong resulting

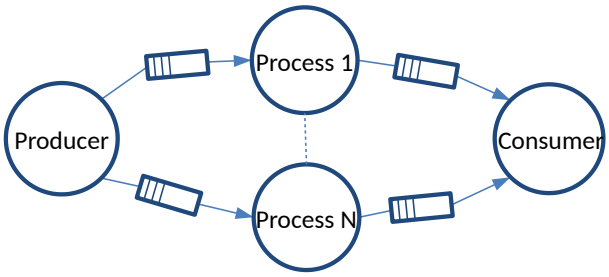


Figure 3 – Example of a simple application parallelized for N cores.

from true sharing can prevent proportional performance scaling, although the hardware provides more resources and processing cores.

Dataflow Model of Computation (MoC)

The general availability of SMPs, ranging from HPC to embedded systems, gives the motivation to explore and enable their benefits for the Dataflow Model of Computation (MoC) [26, 71]. A Model of Computation (MoC) [122] is a high-level representation that enables expressing the specification of a complex application independently from the details of software and hardware implementation. Dataflow is a model of computation used across various domains, including Digital Signal Processing (DSP), video coding, multimedia, telecommunication, computer vision, machine learning, etc. In dataflow, applications are modeled by a graph composed of nodes representing the actors communicating through FIFOs modeled as edges. Each actor implements well-defined actions (functions), consuming input data, processing this data, and producing output data. Each edge is implemented by a FIFO connecting a producer actor output to a consumer actor input. Figure 3 represents an example of a dataflow graph of a producer-consumer application. In this application, *Producer* actor generates and writes data into output FIFOs, which are shared with the actors of $Process_1$ to $Process_N$. Actors $Process_1$ to $Process_N$, reading this data and after processing, write into the output, FIFO shared with the *consumer* actor.

The following features make the dataflow a good candidate to model parallel workloads [26]:

1. *Well-defined communicating interface*: each FIFO can have only one producer and one consumer.
2. *Firing rules*: a set of rules that define when the actor can execute (fire) an action.

In its simplest form, e.g. Synchronous Data Flow (SDF), the actor can start its execution only when all its inputs are available in its respective input FIFOs. The theoretical model considers unbounded FIFOs, which is not possible from a physical implementation point of view. Thus, another rule is that there is enough space in the output FIFOs. These firing rules are explicitly specified in the application. In the context of an SMP, having explicitly these firing rules offers the opportunity to improve the synchronisation of the threads and this is what we show in this thesis.

3. *Non-preemptive processing*: Once the processing part starts, the actor cannot be preempted. From a synchronous point of view, the outputs are instantly produced, whatever the execution time of the action.

From a dataflow specification, code is generated in a given programming language according to the hardware platform. In this thesis, we are interested in multi-threaded C language applications executed on an SMP.

Dataflow applications can potentially lend themselves to expose a fine granularity of parallelism. However, the same feature can directly contribute to increasing data communication and a higher degree of synchronization over shared data (FIFOs) and hence, stress the memory hierarchy when mapped on an SMP by imposing extra data traffic due to coherency. This behavior is specifically studied in this thesis and synchronization in multi-threaded execution of dataflow applications over SMPs is further discussed in the following chapters. Dataflow applications can leverage SMPs, which are prevalent platforms, for their faster execution. On the one hand, dataflow applications mainly rely on producer-consumer data exchange that is simpler than the general form of data sharing in other types of applications. On the other hand, the degree of sharing can be much finer in the dataflow model, leading to data access contention by requiring synchronization at a fine granularity. The Dataflow MoC can break the application to expose a high degree of Task Level Parallelism (TLP) and allow the application developer to exploit the parallel hardware resources more effectively in multi-core SMPs through well-defined producer-consumer communications. This finer granularity of TLP is a double-edged sword. While it enables using a significantly higher number of cores in the SMP, it also requires more synchronizations across many more parallel threads. That is, the synchronization curtails the scalability.

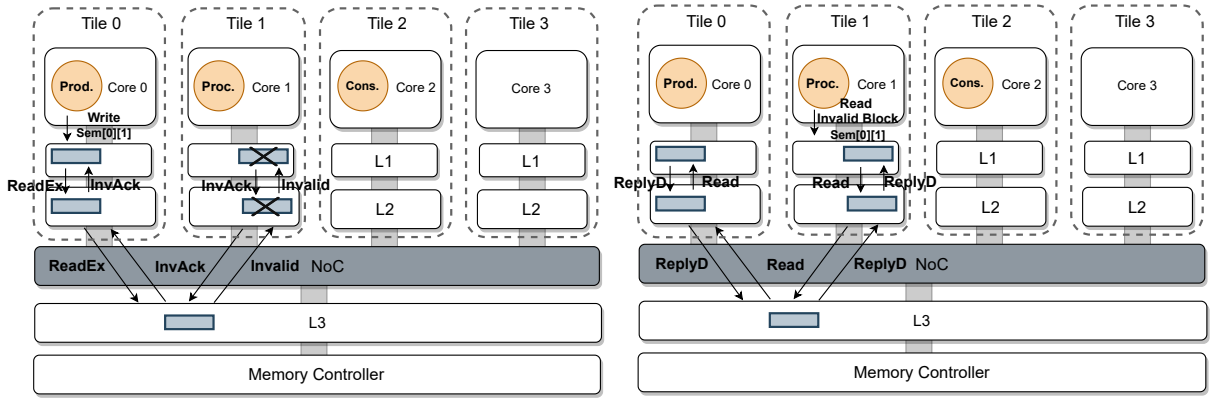
Although frameworks for dataflow applications have been proposed in recent years, they have not fully studied this challenge as the presence of the cache hierarchy in the target architecture is not considered. Thus the optimization methods they provide for

memory management and communication are not well-tailored (well-adapted) to cache hierarchy. The effects of running dataflow application on SMP is not completely explored in the literature. Additionally the current state-of-the-art synchronization methods available on SMP machines, by design, are not suited to the dataflow MoC since they are not aware of the dataflow behavior. Studying these effects and proposing efficient synchronisation mechanisms dedicated to dataflow applications is the main motivation of the work described in this thesis.

Motivation

This motivation is described in more details through the example of a simple dataflow application running on a baseline SMP. As discussed earlier, in a dataflow application, the actors communicate by exchanging the data tokens through FIFOs. Actors wait for the availability of data tokens on their inputs to start execution (checking firing rules). A dataflow framework, responsible for the generation of the code of the application, uses a communication interface to handle the communication between actors.

In a scenario where a dataflow application is running on an SMP, the FIFOs are placed in shared-memory as shared data among the actors, and a synchronization method like semaphore is used as the communication interface to synchronize the actors. When the actors are mapped on different cores, multiple threads access these FIFOs as critical resources. After processing the data and writing them into the FIFO, the producer actor "ups" the semaphore to inform the consumer that the data is ready. On the other side, the consumer actor is waiting for data to be ready then starts execution, so in order to check data availability (checking firing rules), it reads the semaphore state. This synchronization mechanism stresses the memory hierarchy and triggers the cache coherency protocol, resulting in performance overhead. An example helps to discuss this overhead better. Let's consider the example presented in Figure 3, with processing actors executing in parallel on SMP illustrated in Figure 4. Figure 4 illustrates the cache coherency overhead of synchronization of this synthetic application. We assume that the $Producer$, $Processing$, and $Consumer$ actors are mapped on $core_0$, $core_1$ and $core_2$ respectively. In order to synchronize between the actors, a semaphore is dedicated. For this scenario, we have a two-dimension array of semaphores in which each element of this array is dedicated to one synchronization between a source core and destination core of the data exchange (e.g., $semaphore[sourceID][destinationID]$). After $core_0$ finishes execution of $Producer$, it sets



(a) Writer side (b) Reader side

Figure 4 – Illustration of cache coherency overhead related to synchronization of communication

the value of the *semaphore*[0][1] (releasing the lock). As shown in the Figure 4 (a), $core_0$ by writing into the related cache block of the *semaphore*[0][1] switches the state of this block to `Modified` which is done by sending the ReadEx request coherency message [128]. Following this message, the coherency directory residing in L3 sends the invalid request message to the caches containing this cache block as shared data; here the L1 and L2 caches of $core_1$. The caches respond to the invalid request message with an acknowledge message. When $core_1$ wants to check the availability of the data produced by $core_0$ in its input FIFO, it reads the lock, *semaphore*[0][1] value. Figure 4 (b) shows this procedure when $core_1$ reads the cache block containing *semaphore*[0][1] value which is invalidated due to write procedure performed by $core_0$ and, hence encounters a read miss. This read access results in sending read messages down to the coherency directory of L3, which triggers the procedure of updating the cache block with new data of $core_0$ L1 (ReplyD message [128]). Recall that this coherency procedure could cost more than one hundred cycles [141].

In our simple application, by breaking the *Processing* actor into multiple smaller actors with lighter workload executing concurrently, we achieve a finer-grained parallelism in the application as depicted in Figure 3. This representation of the application can exploit the parallel hardware resources in SMP in a more efficient way and can reach higher performance. The total execution time of the synthetic application with n number of *Processing* actors can be defined as follows:

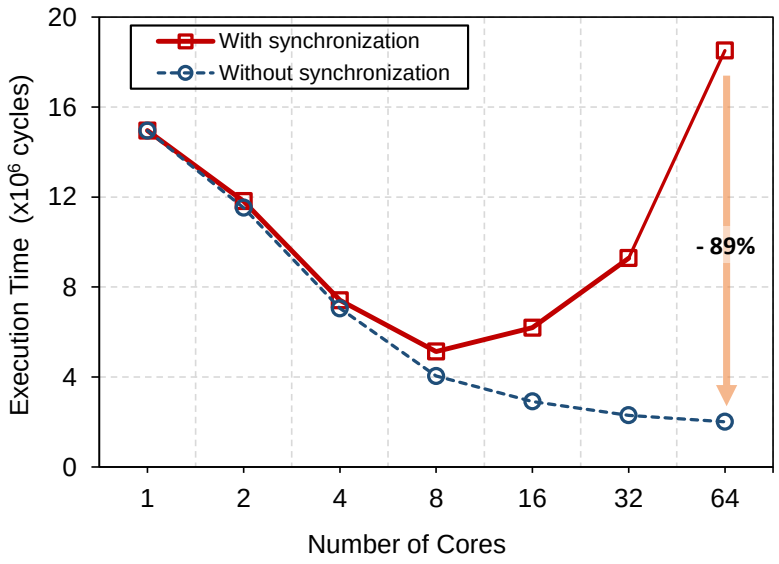


Figure 5 – Communication synchronization impact in a dataflow application.

$$T_{total\ execution\ time} = T_{serial} + T_{parallel}(n) + T_{synchronization}(n) \quad (1)$$

Where:

$T_{serial} = T_{Producer} + T_{Consumer}$

$T_{parallel}(n) = T_{sequential}/n$

$T_{synchronization}(n) = T_{Producer->Processing}(n) + T_{Processing->Consumer}(n)$

The execution time of parallel part of application ($T_{parallel}(n)$) has an inverse relation with the number of threads (n).

Meanwhile, synchronization time of the application has a direct relation with the number of threads. To study the speedup of our application, we run it on an SMP with a varying number of cores, from 1 to 64 cores. Figure 5 shows the result of execution time quantifying the synchronization overhead for our simple dataflow application of Figure 3. Each actor runs in a different core in a dedicated thread. The parallel actors (1 to N in case of Figure 3) are replicated, dividing the same amount of work according to the available number of cores. The figure has two plots, one considers the synchronization overhead, and the another has such overhead deducted from the application execution time. It is possible to observe that the synchronization penalties increase according to the number of cores, representing up to 89% of the application execution time in a scenario with 64 cores.

As explained above, semaphore synchronization causes cache coherency overhead which

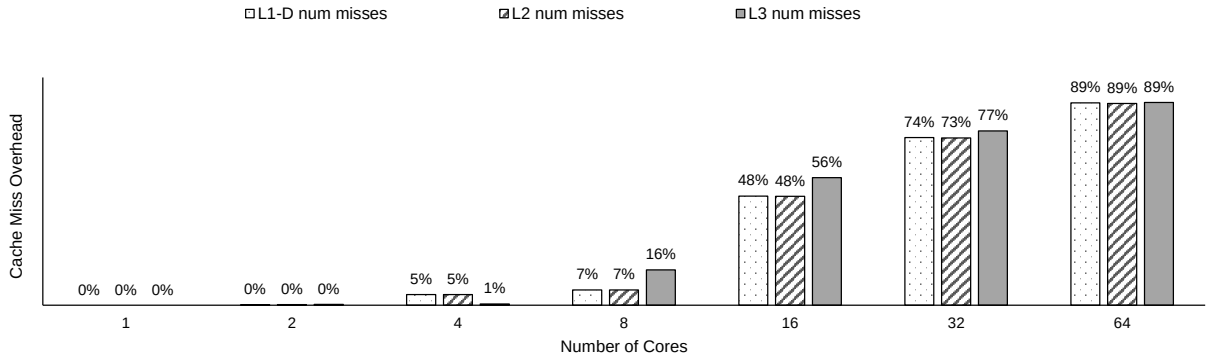
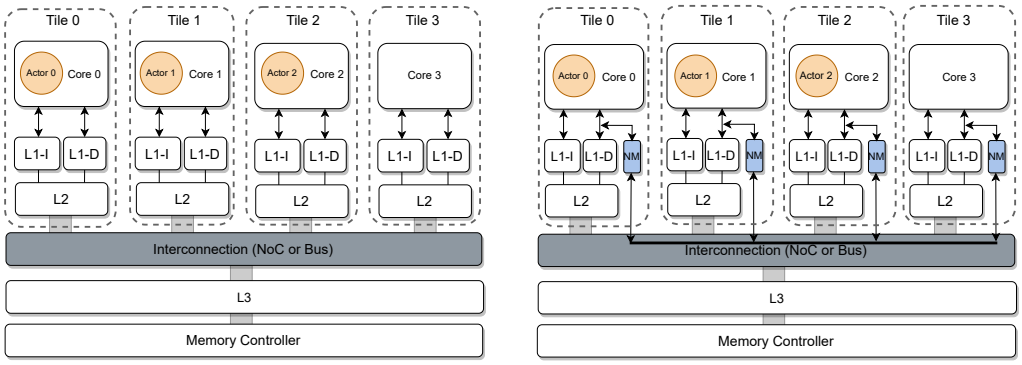


Figure 6 – Cache coherency overhead in the synthetic dataflow application.

decreases the performance. In other words, by increasing the number of cores, cache coherency protocol is more frequently triggered because more synchronization is required. Figure 6 shows the fraction of cache misses caused by semaphore mechanism using the same setup of Figure 5. As the number of cores increases, the overhead of cache misses caused by semaphore also increases.

In summary, the high level of parallelism in dataflow applications helps to efficiently exploit the parallel processors available in an SMP but, as highlighted by the example, the actors' communication imposes significant traffic to the cache hierarchy of SMP because of synchronization. This problem of dataflow applications in SMP systems, which is not properly studied in the literature, is the primary motivation behind our study in this thesis.

The main idea of our proposed technique for synchronization is illustrated in Figure 7. The scenario in Figure 7(a), is depicting a given dataflow application mapped on SMP where the actors are communicating via shared FIFOs based on a typical software synchronization mechanism such as the semaphore provided by the POSIX library. On the right-hand side, Figure 7(b) shows a dataflow-aware hardware logic implemented near the Level-1 (L1) cache which reduces the synchronization time and hence, leads to speed up the communication among the actors in the dataflow application. The proposed logic plays two dedicated roles, one on the producer side and another on consumer side, to handle the synchronization related to data communication in the FIFOs. On the producer side, after the data is written to the FIFO, the logic issues a message as notification to the core that holds the consumer actor. On the consumer side, the module collects all the messages from the producers as a firing rule to inform the waiting consumer actor as soon as its related firing rule is satisfied. The proposed module provides to the cores more than



(a) (b)

Figure 7 – Dataflow application mapped on SMP. (a) With software semaphore synchronization. (b) With NM4SMP synchronization

a pure synchronization by also taking in charge the check of the firing rule. Employing this mechanism avoids the cache coherency traffic resulted from semaphore synchronization. This thesis focuses on developing the logic presented in Figure 7(b). The full stack of the implementation is presented in chapter 4, from software down to hardware level, implemented in a standard baseline SMP architecture. In this stack, a communication interface that uses the proposed synchronization solution is implemented at the software level that employs a driver of the OS level to connect to the hardware logic. In brief, this contribution helps in making the SMP dataflow-friendly.

Thesis Organization

The two main contributions of this thesis are presented in two different chapters and are summarized as follows.

1. Cache evaluation and memory management techniques: This first contribution contains two parts. At first we perform a comprehensive study of the behavior of dataflow applications on SMP from a cache hierarchy point of view. This kind of in-depth study was missing in the literature. A generally accepted idea is that bigger and multi-level caches improve the performance of applications. This work evaluates such a hypothesis in a broad experiment campaign adopting different multi-core configurations related to the number of cores and cache parameters (size, sharing, controllers). The results show that bigger is not always better, and the foreseen future of more cores and bigger caches do

not guarantee software-free better performance for dataflow applications. Secondly, the memory management optimization methods employed by the current frameworks do not generally exploit the cache hierarchy of the shared-memory multiprocessors efficiently for dataflow applications. For instance, PREESM framework uses exclusively `memcpy()` for *Broadcast* actors which results in the eviction of useful data in the cache. This is known as cache thrashing. In this thesis, we investigate the adoption of two existing memory management strategies for dataflow applications: Copy-on-Write (CoW) and Non-Temporal Memory (NTM). These techniques allow to bypass the cache hierarchy and reduce the cache thrashing. We show through experiments that in some cases, these techniques can optimize the data movement in the cache hierarchy and improve the performance.

2. Notifying Memory for SMP (NM4SMP) for dataflow communication: The communication interface relying on software semaphores used by existing dataflow frameworks has cache coherency overhead that penalizes the performance of the application running on an SMP. In this thesis we provide a novel communication interface using a near-memory processing implementation. Our *contributions* in this section are summarized as:

- A HW/SW co-design, including library, driver, and a near-memory hardware on-chip device to synchronize dataflow actors, called NM4SMP (Notifying Memory for SMP).
- A methodology to integrate the NM4SMP co-design in the process of dataflow modeling, code generation and compilation of reconfigurable and static applications.
- A detailed power and performance evaluation of static and reconfigurable dataflow applications using NM4SMP for generic and low-power processors.

The rest of this document is organized as follows:

- Chapter 1 provides detailed background on multiprocessor architectures, memory hierarchy, synchronisation and the simulator used for our experiments.
- Chapter 2 presents Dataflow MoCs and rapid prototyping tools, with an emphasis on the existing tools used for our experiments: PREESM and the SPiDER runtime management layer.
- Chapter 3 presents the first contribution of this thesis: the behavior of dataflow applications on SMP and the use of existing dynamic memory management tech-

niques.

- Chapter 4 presents the second contribution of this thesis: Notifying Memory for Shared-memory Multiprocessor (NM4SMP) for dataflow applications. We propose NM4SMP for static dataflow applications as well as reconfigurable dataflow applications that are managed by a runtime management layer. We also present the evaluation and results of our approach.

Chapter 1

Symmetric Shared-memory Multiprocessor (SMP)

This chapter provides the background required to understand the main architectural and related synchronization concepts addressed in this thesis. It is divided in four sections. The first one discusses shared-memory processors (SMP) and delves into the various aspects of the architecture that impacts the correctness and performance of execution of applications. The second one focuses on the synchronisation mechanisms available to allow the safe sharing of data between concurrent threads. Then we come back on the main concepts related to the caches used to speed up access to potentially shared data. Finally, we explore the multicore simulators that are used to evaluate the proposed solutions in SMPs, and evaluate the simulator we utilized for our research in this thesis.

1.1 Symmetric Shared-memory Multiprocessor (SMP)

Symmetric Shared-memory Multiprocessor (SMP) is the highly dominant solution for current processor chips. SMP contains two or more processing elements (cores) sharing the same unique logical memory address space and is generally managed by one Operating System (OS) [116, 42]. An SMP architecture includes a single chip that comprises multiple cores which are connected to a physically shared main memory. Although a SMP might be designed without any cache, a commercial SMP is now built upon shared caching subsystem [116, 42]. This kind of architecture is the starting point of this thesis, and illustrated in Figure 1.1. It is composed of several tiles. The tiles are homogeneous, with the same hardware architecture. Figure 1.1 shows an example of a cache with three levels. Levels 1 and 2 are embedded inside the tile, working as private cache memories. The last-level cache (LLC), level 3, is shared by all the tiles and also interfaces with a memory controller. The memory controller interfaces with an off-chip main memory. The SMP assumes a shared memory that implements a cache-coherence protocol. Such a

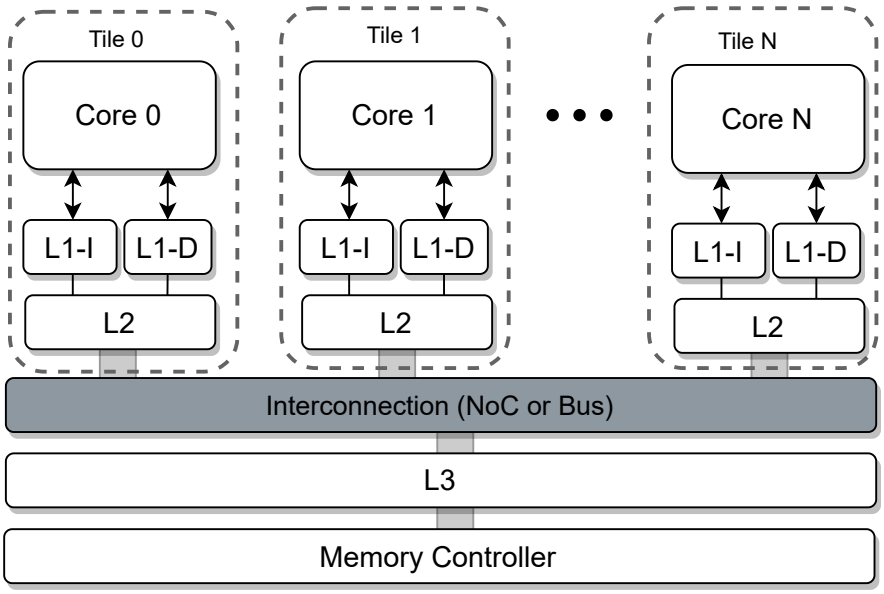


Figure 1.1 – Reference multi-core with symmetric processing (SMP) assumed in this work.

cache organization is common in commercial products, it is for instance adopted in the SMP architectures used for our experiments in this thesis (Intel Xeon and Atom) [65]. Subsection 1.3 enters in details about cache mechanisms.

The tiles are interconnected by a bus or by a Network-on-Chip (NoC). A NoC is an on-chip communication infrastructure to interconnect tiles, decoupling the computation from communication. The NoC structure contains routers and wires. Routers have the function to implement the network control logic, defining the path for each exchanged packet between a source and target tile. Wires have the function to interconnect routers and to connect each router with its local tiles. While buses are typically used in multi-core with low core counts, NoCs are preferred in many-core architectures due to their benefits against the bus, as parallel communication among tiles and scalability [100, 19].

1.2 Synchronization for SMP

Our work focuses on providing specialized hardware mechanisms for synchronisation in dataflow applications. As such, it is related to previously proposed synchronization mechanisms that are surveyed in this section, which were (and still are) studied independently of the number of cores or the use of caches. The synchronisation issue is further exacerbated by the use of caches, and is specifically discussed in the next section.

The advantage of shared-memory is that the application developer sees the memory as one, and all the cores can access all the memory locations. An important problem is to *synchronize* the shared memory accesses among different threads running in parallel. The synchronization means that threads must find an *agreement* with other threads to exclusively access to the shared data. The region of the thread program that accesses the shared data is called critical section. Critical sections are synchronized by ensuring a mutual exclusion access between the threads. The system used to protect the critical region is called a *lock*. Note that ultimately all software-based mutual exclusion lock algorithms are based on Atomic Memory Operations (AMO) at hardware level [108]. AMO means that the memory operation performed by the core is indivisible; therefore, no other entry can slip or suspend the operation involved in the AMO while the AMO was not finished.

According to the number of independent processes that compete for the same critical resource, and their frequency of access, it is possible to measure the *contention*. The contention occurs whenever one process or thread attempts to acquire a lock held by another process or thread. The finer-grained the available locks, the less likely one process/thread will request a lock held by the other. For example, locking a row rather than the entire table or locking a cell rather than the entire row will be less prone to contention.

In the next sections, we provide a comprehensive survey of studies on synchronization approaches at different abstraction levels. A lot of optimized synchronization solutions have been proposed by the researchers at software [95, 125, 36, 10, 54, 135], ISA [146, 139] and hardware level [80, 141, 46, 98, 132, 34, 47, 1, 3, 2, 81, 103, 84, 138, 149, 52, 69, 92].

1.2.1 Synchronization Support in Software

This section introduces the main concepts related to software synchronisation, using the example of the most adopted techniques available in the main stream OS, including Linux.

Spin-lock

Spin-lock is the most basic form to implement a lock. In summary, the threads perform a loop that constantly checks the value of a lock variable, shared among all cores. This checking is performed by AMO. A spin-lock is a lock which causes a thread trying to acquire it to simply wait in a loop ("spin") while repeatedly checking whether the lock

Algorithm 1: Example of spin-lock used in RISC-V

```
1 /* RISC_V spin_lock and unlock implementations */
2 void spinlock(lock *lock_value){
3 bool locked;
4 //Spin lock
5 do {
6     locked = CAS(lock_value, 1, 0);
7     }while(!locked);
8 }
```

is available through Read-Modify-Write (RMW) operation [9]. Since the thread remains active but is not performing a useful task, the use of such a lock results in a busy waiting. Once acquired, spin-locks will usually be held until they are explicitly released.

Conventional designs employ either busy waiting or blocking for waiting. With busy waiting, waiting threads remain active, polling the lock until they manage to acquire it. With blocking, waiting threads release their hardware context to the OS. The OS is in charge of unblocking these "sleeping" threads when the owner releases that lock [10].

Algorithm 1 presents a function using `CAS` (`Compare And Swap`) instruction which is the basic implementation of atomic RMW lock operation, to acquire the lock in a RISC-V processor. When a thread calls this function it enters a while loop and in `CAS`, it compares the lock value with 1 if it is equal it means that the lock is successfully caught then swap it to 0 and returns `True`. If the lock value is equal to 0 it means that the lock is caught by another thread and `CAS` returns `False` so the thread should try again.

Test-and-Set (TAS), Test-and-Test-and-Set (TTAS), and ticket lock [95] (TICKET) are spinlock types in which threads spin the locks on a single memory location in busy waiting mode. Because of the simplicity of spinlocks, they are fast; however, it is a mechanism that will lead to higher degrees of contention when multiple threads are competing to acquire the lock. The main downside with spin lock is that it leads to three problems: (1) the atomic spin-lock operations such like `compare_and_swap` is by definition non-preemptible, and waiting for the operation to perform degrades the performance as the number of competing processors increases; (2) In multi-threaded systems, a thread that is preempted during its critical section (meaning that it keeps the lock and no other thread can access to the shared resource) can delay every other threads that needs to acquire the lock [144]. This is typically known as the inversion of priorities. (3) It is noticeable that waiting processes stress the cache by spinning the lock which is energy consuming as they

need cache consistency and messages in coherency protocol [129]. In proposed efficient lock techniques, an Operating System as Linux, put the waiting threads on sleep instead of staying in a busy waiting state and constantly spinning the lock.

Mutexes and Semaphores

Mutexes: Mutex or Mutual Exclusion Object provides access to a shared resource so that all the processes use the resource, but only one process is allowed to use the resource at a time. Mutex uses the lock-based technique to handle the critical section problem. The system will generate a mutex object with a unique name or ID whenever a process requests access. So, as the process wants to use that resource, it holds a lock on the object. After locking, the process uses the resource and lastly releases the mutex object. By locking the object, that shared resource is allocated to that particular process, and no other process can take the resource. Hence, in the critical section, no other processes are allowed to use the shared resource.

Semaphores: In semaphore synchronization technique, threads utilize two functions to modify the semaphore value `wait()` and `signal()`, but the functions allow only one thread to change the value at a specific time, i.e., two threads cannot simultaneously change the semaphore value. There are two types of semaphores; **Counting semaphores** and **Binary semaphores**.

In **Counting semaphores**, at first, the semaphore value is initialized with the number of resources available. Afterward, whenever a process demands some resource, the *wait()* function is called, and the semaphore value is decremented. The process then uses the resource, and after using the resource, it increments the semaphore value by calling *signal()* function. So, as semaphore value becomes 0, it means that all the shared resources are being used by the processes and there is no resource left to be used. Any other processes demand the resources, then they need to wait.

In **Binary semaphores**, the semaphore value can be 0 or 1 and it is initialized with 1. When one process wants to use some resources, it calls the *wait()* function and sets the semaphore value to 0. It uses the resource, and then by calling the *signal()* function and setting the semaphore value to 1, it releases the resource. When the semaphore value is 0, and a process wants to use the same resource, the process needs to wait for it to be released by the previous process.

Semaphores and Mutexes are both sleep-wait spin-lock, i.e., while a process or thread

can not take the lock (Mutex) or is waiting for the semaphore value to become positive, the operating system puts the thread on sleep.

The differences between mutex and semaphore are as follows: as Mutexes employ a locking mechanism for synchronization, the semaphore techniques employ a wait and signal mechanism. In synchronization using mutexes, multiple process can access one shared resource but only one process at a time. The semaphore technique also allows multiple processes to access the finite number of the resource until some resources become free. In mutex mechanism, the lock can be acquired and released by the same process. Conversely, the semaphore value can be changed by any process that needs some resource, but only one process can modify the value at a time.

Parallel Application Programming Libraries

Parallel application programming libraries are used to ensure a transparent and clear programming method to parallel application developers. They contains a set of Application Programming Interface (API) focused on allowing the developer to model the application following a parallel paradigm. The developer can call such API functions of programs in the code, allowing the OS to dynamically create new threads and to manage its execution among different cores.

POSIX Threads (Pthreads) libraries are standard-based thread APIs for C/C++, supported in Linux. It allows one to create a new parallel process flow. Pthread is the most efficient on multiprocessor systems where the process flow can be scheduled to be executed on another processor to gain parallelism to achieve performance. Threads require less overhead than "forking" or creating a new process because the system does not initialize a new virtual memory space and environment for the process. While most effective on a multiprocessor system, Pthreads benefits are also noticed on uniprocessor systems. Uniprocessors exploit latency in I/O and other system functions that may stop process execution, i.e., one thread may execute while another is waiting for I/O or some other system latency. All threads from the process share the same address space. A thread is created by defining a function and its arguments which will be processed in the thread. The main goal behind using the POSIX thread library in the program is to speed up its execution. Some parallel programming approaches like OpenMP rely on Pthread library.

Queue-based synchronization

Another group of synchronization techniques is based on queues. The queue-based techniques can mitigate the contention problem of the spinlock approach. These approaches substitute a single lock variable (or the boolean flag of a test and set lock) with a queue of waiting threads. Each thread knows its own place in line: it waits for its predecessor to finish before entering the critical section and signals its successor when it is done [125]. To reduce the contention problem, Mellor-Crummey and Scott [125] showed how to adapt queued spinlocks to the reader-writer case. Specifically, they presented reader-preference, writer-preference, and fair reader-writer locks based on the MCS lock.

MCS [125] and CLH [36] are queue-based spinlocks that create a queue of waiting threads where each thread is spinning on a unique location when busy waiting. Thus queue-based locks solve the problem of the single-memory-location bottleneck of simple spinlocks.

Some solutions combine multiple lock types to benefit from their advantages in specific situations. For instance, Antic et al. [10], introduce GLS, a middleware that dynamically switches among three locking algorithms (Ticket, MCS [125], Pthread mutex). It uses Ticket at low contention levels, MCS at high contention levels, and Pthread when it detects over-threading (i.e., more threads than cores).

A recent comprehensive analysis on software synchronization methods is presented by Guerraoui et al. [54]. Their research performs a thorough study considering throughput (traditionally the primary performance metric), energy efficiency, and tail latency of 28 state-of-the-art mutex lock algorithms, on 40 applications, on four different multicore machines. They have achieved significant findings including: i) rather than lock/unlock interface, applications stress the full locking API (e.g., trylocks, condition variables), (ii) application performance can be directly affected by the memory footprint of a lock, (iii) the interaction between locks and scheduling is a significant factor for performance (iv) no lock algorithm is systematically the best, (v) it is difficult to choose the best lock, and (vi) in the context of lock algorithms, energy efficiency has a direct relation with throughput.

To the best of our knowledge, the approach introduced by Szustack [135] is the only proposed software synchronization solution regarding dataflow applications for shared memory multicore architectures. This work introduces a strategy to speedup the thread synchronization for dataflow applications at the software level. The approach consists of synchronizing interdependent threads only by grouping them at the software level,

instead of using global barriers. Interdependent threads have producer-consumer data dependencies. This approach is evaluated over two Stencil-based dataflow applications. Stencil is a class of numerical data processing solution which updates array elements according to some fixed patterns. In this algorithm, a single synchronization point called is used to synchronize a group of interdependent threads. This synchronization point uses a counter that contains the number of threads in the group and a global release flag. Once one of the threads in the group arrives to the synchronization point, the counter is decremented and the thread is suspended until the global flag is released. The flag is only released when all the threads have arrived and the counter has reached zero. A separate flag is used to release all the waiting threads. The synchronisation model proposed is limited to stencil style of programming and additionally it is still a centralized software-based mechanism that is prone to contention between the threads. It is not a viable solution for our problem, which needs a scalable solution. Our proposition, in contrast, considers pairs of producer-consumers and is not centralized.

1.2.2 Synchronization Support in the ISA

Contrary to the software synchronization approaches in which typically a thread or a process is spinning on a lock value, some processors propose event-based mechanisms for synchronization in their Instruction-Set Architecture (ISA).

Among commercial ISAs, ARM instruction-set [146] includes several instructions for synchronization, Set Event (`SEV`), Wait For Event (`WFE`), Wait for Interrupt (`WFI`), and Yield. The `SEV` causes an event to signal to all cores within a multiprocessor system. The event is monitored by the `WFE` instruction through an Event Register. Intel ISAs offer the `XCHG` instruction to compare and set locks and the `TEST` for checking the status of the locks without changing its value.

In a different vein, the VAX instruction-set [139] supports hardware-accelerated queues. It has different instructions for communication on the hardware queue. The VAX instructions `INSQHI`, `INSQTI`, `REMQHI`, and `REMQTI` allow insertion or removal at the head or tail of a self-relative double-linked list. These instructions are interlocked and cannot be interrupted; this feature stops other processors from updating the queue at the same time. These complexities make it complicated to reach high performance, and programmers often prefer to implement their own software queues [80].

1.2.3 Synchronization Support in Hardware

In this section, we overview the existing synchronization solutions in hardware. The studied hardware synchronization approaches can be categorized into different groups according to the techniques employed to accelerate the synchronization. The techniques include: (1) employing queue-based approach (2) accommodating the synchronization primitives in local memories or network interface to avoid shared memory accesses (3) bypassing the shared memory using a dedicated network for synchronization messages.

Queue-based Synchronization

In the first group, synchronization solutions primarily employ a queuing mechanism for synchronization approach [80, 141, 69]. Using queues are common in data communication and synchronization approaches of parallel programs. The **Q-based** approaches are efficient for the parallelism that are coarse-grained enough to amortize the enqueue and dequeue cost [80]. The hardware Q-based approaches are introduced to reduce the overhead of the software Q-based communications related to enqueue and dequeue operations.

As managing software queues in synchronization of fine-grain parallelism is expensive [80], several works introduced hardware synchronization approaches focus on hardware queues to decrease the overhead [80, 141, 69, 138]. However, they require ISA modification and custom interconnect. Additionally, they impose burdens on the operating system in terms of preserving the states of the queues across context switches.

The Hardware-Accelerated Queue (HAQu) [80] takes advantage of the benefits of hardware queuing while maintaining the benefits of software queues. Their proposed design provides three fundamental features: (1) it supplies a single instruction for *enqueuing/dequeuing* which decreases the overhead of fine-grained communications; (2) HAQu leverages on-chip interconnect and cache coherence protocol for transferring data from producer to consumer; (3) it solves the problem of context switch overhead of the OS. In HAQu, the queues states are entirely stored in the virtual address space of the application. With the help of these features, HAQu solution provides higher throughput and lower latency compared to optimized software implementations. Communication Acceleration Framework (CAF) [141], similar to HAQu, introduces a co-optimized software and hardware solution to accelerate the hardware queue operations. In CAF, they offload the communication responsibilities from processor and memory to a dedicated hardware device, Queue Management Device (QMD), attached to the Network on Chip to improve core communi-

cation. Contrary to HAQu, CAF does not impose memory access and coherency overhead. Moreover, it has less core modification than HAQu (only three new instructions added to ISA).

Swarm [69] is a scalable tiled multicore architecture designed for the applications presenting the *ordered irregular parallelism* in their execution pattern. Swarm architecture employs implicit communication, which is a commit protocol using hardware task queues. They use hardware queues for their proposed scheduler to synchronize the tasks. Applications with *ordered irregular parallelism* can be characterized by three main features [111, 59]. (i) Each task is an event (ii) which starts execution at specific time and modifies a specific component. (iii) Tasks dynamically create other tasks. This parallelism is abundant in many domains like simulation, graph analytics, and databases.

The Swarm execution model consists of distributed task queues, speculative out-of-order task execution, and ordered task commits, which helps Swarm's micro-architecture scale efficiently.

Vallejo et al. [138] introduce a Lock Control Unit (LCU) with a Local Reservation Table (LRT) that manages the hardware locks queues. LCU is attached to the core to accelerate reader-writer locking for core-to-core transfers. The authors assign the hardware locks to the lock requester by associating the thread ID of the lock requester to the lock. By doing so, the locks are decoupled from the cores and hence, provide efficient threads migration.

Accommodating Synchronization Primitives

Since memory access in synchronization operations is always a bottleneck, in order to accelerate synchronization, the researchers try to avoid shared memory access. The second group of hardware synchronization approaches are the solutions in which the synchronization resources are accommodated in local memory of Network Interface [46, 98, 47, 84, 2, 3, 81, 103].

Storing the lock values in a dedicated local memory makes synchronization accesses faster rather than when they are placed in shared memory. This approach is the inspiration of the works presented in [46, 98, 47, 34]. Ferri et al. [46] introduce a distributed hardware semaphore synchronization where cores spin on the lock value in their local Scratched Pad Memory (SPM) instead of shared memory. By offloading semaphore accesses from the system bus to a dedicated local and shared-memory for semaphores, this approach provides performance improvement and energy efficiency.

Monchiero et al. [98] present a solution based on the idea of managing the synchronization operation requests (spinlocks and events) locally using a hardware block called Synchronization-operation Buffer (SB) in the memory controller. Based on the fact that the synchronization primitives can create considerable memory and interconnect contention, the SB technique uses first–in first–out (FIFO) to order the pending lock requests in a queue. By employing this ordering approach they prevent the lock starvation. Besides, this approach by monitoring the contended shared variable avoids the contention. The approach proposed by France-Pillois et al. [47] employs a decentralized hardware synchronization solution for MPSoCs. They provided a dedicated memory for dynamically automatic re-homing of the locks to benefit from spatial locality. This approach promotes the lock access time and decreases overall access latency and network traffic resulted by synchronization accesses.

Giannoula et al. [52] propose synchronization solution for Near-Data-Processing systems and propose an end-to-end (message-passing) synchronization solution called SynCron. In their approach each NDP has a local synchronization management unit which passing the synchronization messages to the master synchronization management unit in another NDP to handle it. The master management unit manages all the locks using a table which contains all the lock information of the system.

Zhu et al. present a Synchronization State Buffer (SSB) for fine-grain inter-thread synchronization. SSB is a scalable hardware design attached to the memory controller of each memory bank [149].

Dedicated Interconnection for Synchronization Primitives

The second approach to reduces memory overhead of synchronization is bypassing the memory by dedicating specific interconnect for handling the synchronization.

Abellan et al. in [3] propose a fast and efficient dedicated network for barrier synchronization and highly-contended locks. They have designed a simple and scalable G-line-based network that operates independently of the main data network, and that is aimed at carrying out barrier synchronizations efficiently. In another work of the same team [2], they propose and evaluate GLocks, a hardware-supported implementation for highly-contended locks in the context of many-core CMPs. GLocks use a token-based message-passing protocol over a dedicated network built on state-of-the-art technology. This approach skips the memory hierarchy to provide a non-intrusive, extremely efficient and fair lock implementation with negligible impact on energy consumption or die area.

CM5 [81] provides a barrier primitive via a dedicated physical network. TLSync [103] employs the high-frequency part of the spectrum in a transmission-line broadcast network for implementing the hardware barrier. In [1] a global fine-grain synchronization using low latency on-chip wireless communication called WiSync is introduced for shared-memory architectures.

Besides all above reviewed approaches, the solution introduced by Cataldo et al., [34] benefits from both two techniques; using local memory for the locks and handling the synchronization operations in interconnection networks to reduce the shared memory access overhead. They propose a HW/SW co-design architecture called *Subutai* that handles all the Pthread synchronization primitives (mutex, condition, barrier) in the network interface. Moreover, *Subutai* uses a local Scratched Pad Memory to store the synchronization primitives and queues to record threads waiting for new events on the primitives. Moreover, they have employ a dynamic technique for memory usage in which a double-linked list of events is employed to dynamically allocate the queue entries which allows Subutai to consume memory on demand.

Contrary to the approaches based on avoiding shared memory accesses for synchronization, Liang et al. [84] introduce a solution in which the synchronization primitives are placed in the shared memory. Their proposed approach called MISAR is a distributed hardware synchronization accelerator for tiled multiprocessors chips. It includes hardware locks called MSAs (Minimalistic Synchronization Accelerator) which supports all thread synchronization types (locks, barrier and condition variables) for tiles along with an Overflow Management Unit (OMU). MSA of each tile which contains all the locks of the tile is placed in the LLC of the tile. The OMU dynamically switches to the software exception handler when the number of synchronization variables are much larger than the number of hardware synchronization resources. They have also proposed ISA extensions to facilitate adopting their proposed synchronization library.

1.2.4 Qualitative Comparison

This dissertation exclusively focuses on providing a specialized hardware synchronization mechanism for dataflow application. The proposed synchronization mechanism, named NM4SMP, is specifically focused on the dataflow MoC and compliant with SMP architecture and programming models. It is thus different to all generic solutions previously mentioned and to *Notifying Memory* concept, already developed in the team, which is Dataflow-specific but for a distributed memory architecture without cache memory.

Table 1.1 – Related works addressing data synchronization for parallel applications.

	Solution	Designed for	ISA Ext.	Q-based	Spin-Wait	Direct Notif.	Overflow Management	Scalability
Software	TICKET [95]	Parallel applications	×	×	✓	×	–	×
	MCS [125]	Parallel applications	×	✓	✓	×	–	good
	CLH [36]	Parallel applications	×	✓	✓	×	–	good
	GLS [10]	Parallel applications	×	✓*	✓*	×	–	×
	Szustak [135]	Dataflow applications with stencil computation	×	×	✓	×	–	–
ISA	ARM [146]	Parallel applications	–	×	×	✓	–	✓
	VAX [139]	Parallel applications	–	✓	×	✓	–	✓
Hardware	HAQu [80]	Applications with Q-based communication	✓	✓	–	–	×	Cond.
	CAF [141]	Applications with Q-based communication	✓	✓	–	–	×	good
	LCU [138]	Parallel applications	✓	✓	✓	✓	✓ Partially integrated	good
	SB [98]	Parallel applications	×	×	✓	✓	×	good
	MISAR [84]	Parallel applications	✓	✓	×	✓	✓ Handled by programmer	good
	SSB [149]	Parallel applications	×	×	✓	×	✓ Partially integrated	single-chip
	Swarm [69]	Applications with ordered irregular parallelism	✓	✓	×	–	✓ Fully integrated	good
	NM [92]	Dataflow applications	×	×	×	✓	–	good
	NM4SMP	Dataflow applications	×	×	×	✓	✓	good

In this context, this section provides a qualitative comparison of the related works and position our proposed solution, NM4SMP, among them. Table 1.1 summarizes the existing approaches based on the prevalent metrics of communication methods ranging from implementation metrics (legacy code, implementation cost, etc.) to the parameters related to the performance (Spin-Wait, Overflow management, etc.). At the end of the section we perform a comparison and differentiation of our method with these approaches.

The column **Designed for** details the application type that the work is targeting. As can be seen, most of the approaches are provided to support Pthread-based parallel applications [80, 142, 95, 125, 36, 10, 146, 139, 138, 98, 84, 149].

The specific microarchitecture Swarm [69] supports the applications with *ordered irregular parallelism* in their computation pattern. *Ordered irregular parallelism* is abundant in many domains, such as simulation, graph analytics, and databases. In this applications each task is an event (e.g., executing an instruction in a simulated core). Each task needs to run at a specific simulated time (introducing order constraints among tasks), and modifies a specific component (possibly introducing data dependencies among tasks). Tasks dynamically create other tasks (e.g., a simulated memory access), possibly for other components (e.g., a simulated cache), and schedule them for a future simulated time.

To the best of our knowledge, there exists only two works proposed for dataflow applications: (1) a software synchronization solution presented by Szustak et al. [135] designed for the dataflow applications with stencil computation, (2) a hardware synchronization solution introduced by Martin et al., called Notifying Memory [92] for dynamic dataflow

applications. The Notifying Memory approach laid the foundations of this thesis, but the first prototype was implemented in the network interface of a network-on-chip of a distributed shared memory architecture, with no cache. The experiments in their work are performed for a single application (MPEG4-SP) whereas we study several applications, with different behaviors and characteristics. Moreover, the results focus only on the traffic at network level, whereas we consider the whole platform, including the impact on the execution time. Finally, we implement the solution in the L1 cache of a commercial centralized shared memory architecture with a cache hierarchy, which raises new challenges, like a close coupling to the processor and a controller that deals with the notifier and listener simultaneously.

The column **ISA Ext.** classifies the works that require extending the ISA to implement the proposed technique. In some hardware approaches, HAQu [80], CAF [141], LCU [138], MISAR [84], Swarm [69], an extension to the existing ISA is required to exploit the underlying hardware. Hence, this feature is solely related to the hardware solutions. Moreover, adding new instructions requires modifications to the decoder of processors which adds to the implementation cost [108]. Among the solutions, HAQu [80], CAF [141],LCU [138], MISAR [84] and Swarm [69] have ISA extension while SB [98], SSB [149], Notifying Memory [92] and our proposed solution, NM4SMP, do not impose any ISA extension overhead.

The column **Q-based** classifies the works that provide optimisations targeting to use queue-based communication. The HAQu [80], CAF [141], LCU [138] and Swarm [69] employ the queue-based synchronization approaches. As mentioned, queue-based techniques are not attractive for fine-grain parallelism due to expensive enqueuing and dequeuing operations. Being highly vulnerable to preemption is a hidden draw-back of queue-based techniques [60]. The feature **Spin-Wait** classifies the works targeting spin-wait synchronization. A spin-wait method performs constant synchronization retries. Typically, majority of the synchronizations with remote atomic primitives, read-modify-write (RMW) operations, require spin-wait mechanism which provokes high energy consumption due to incurring traffic across NoC and memory hierarchy to ensure the consistency of the lock. Additionally, these spin-wait locks do not scale beyond a modest number of processors [60]. As presented in the table, all of the software approaches employ spin-wait for acquiring the synchronization value [95, 125, 36, 10, 54, 135]. In ISA solutions, ARM [146] and VAX [139] architectures, specific instructions are provided to interrupt the waiting core and hence it does not need busy-waiting as they handle it via event and interrupt.

Among hardware solutions, the proposed approaches, LCU [138], SB [98] and SSB [149],

employ spin-wait in local memory technique to avoid cache coherency overhead. Our proposed approach does not need spin-wait and even in local memory we check whenever the local memory is updated.

The column **Direct Notification** classifies works that employ the technique of Direct Notification. In such a technique, a direct notification message is sent to only the waiting core when the synchronization variable becomes available. This mechanism minimizes the traffic involved upon a release operation [52]. The ISA and hardware approaches employ a notifying mechanism rather than spin-wait to avoid the resulting traffic of spin-wait on memory hierarchy and NoC. Hardware implementation of the direct notification usually requires a high cost of implementation, for instance, a dedicated interconnect network for the notification message. Hence, some approaches, SB [98] and SSB [149], benefit both techniques which is notification-based aligned with spin-wait based. MISAR [84], LCU [138] and Notifying Memory (NM) [92] employ a notification-based mechanism to directly notify the waiting core. MISAR as well as other approaches provides thread level synchronization. Our approach is also a direct-notification-based technique that supports dataflow applications.

The column **Overflow Management** classifies works that support overflow management. The overflow management is a feature needed specifically for the solutions implemented in hardware. Due to limited hardware resources in a hardware communication mechanism, a resource overflow manager is required to guarantee the correctness of the application execution when synchronizations tracking are exceeding the hardware structures (memory, buffers) dedicated to synchronization.

SSB [149] and LCU manage the overflow of hardware lock resources using a preallocated table in main memory and at overflow time they switch to software exception handlers which impose a large overhead due to the OS intervention [52]. MiSAR [84] requires the programmer to manage the overflow scenarios by switching to the software synchronization alternatives (e.g., pthread library provided by the OS). In SWARM [69] also when the task queues are fulled by removing the non-speculative tasks from the queue and storing them in memory to be re-enqueued later it gives the illusion of of unbounded hardware task queues [68, 75, 120]. Our approach to handle the overflow is similar to MISAR approach and we switch to the default software synchronization library for the synchronizations that exceed the limited resources. For static dataflow applications as the behavior of the application is anticipated at compile time, overflow is handled at compile time, hence it does not impose any overhead. For reconfigurable applications, dynamically

at runtime, with help of runtime manager, software synchronization will be employed as the hardware lock resources overflow.

The last column of Table 1.1 classifies the works according to **Scalability**, referring as the capability to keep performance level when the number of communication requests increase [149]. With a large number of cores, due to low scalability of conventional hardware coherence protocols [61, 72, 73, 129] the coherence-based synchronization approaches perform unsuccessfully. Single-line locks result in coherent contention and scale poorly, while queue-based approaches solve problem and scale very well [138]. As can be seen in the table, among the software solutions, which are typically lock based, only two solutions, MSC [96] and CLH [36], have good scalability and most of the queue based hardware approaches are scalable. However, scalability is also critically restricted by the long transfer latency and low bandwidth of the interconnect used between the cores [52] as HAQu [80] is able to scale-up and achieve high throughput on next generation interconnects. Our communication solution is not coherency based approach, that is, it does not use coherency protocol for lock transfer. Instead, it employs direct notification as synchronization variable, thus, by increasing the number of synchronizations due to scaling the system NM4SMP does not suffer from the cache hierarchy overhead and hence, provides good scalability.

1.3 Caching: terminology, mechanisms, and synchronisation

The focus of this document is on physically shared memory architectures that use caching. Caching has been intensively studied during the last decades (and still is). This section does not go into the details of the most advanced techniques but introduces the terminology and the main concepts to understand the contributions of this thesis.

A good way to understand the cache concept is to use the library's example from Patterson's book [108]. Imagine a student seated at her/his desk inside a library. While she/he is studying, some books need to be brought and placed on the desk to make a quick consult. While the effort to read a book from the desk is very low, the space of the desk is limited, and the student can not have all the books on the desk at the same time. Therefore, it is necessary to return to the shelves, pick the required book, and bring it to the desk to read. As the research of the student progresses, more books are needed to bring to the desk, and the space of the desk becomes limited. So the student needs to

return some books back to the shelves to open space for the new ones. It is important to keep the books more accessible on the desk and maybe, if the desk has free space, bring some additional books, closely related to the subject, imagining that they will be referred soon.

This famous example quickly introduces some fundamental concepts of cache and, at the same time, its challenges. The student can be considered as the core processor. The books are considered as data blocks, and the library can be considered as the main memory which is located off-chip. The blocks are the smallest unit of data transferring between the cache and main memory. The desk of the student can be considered as L1-cache of Figure 1.1. The L2 cache is as mid-level cache (MLC) if the student decided to put an additional desk next to the main desk. The access to L2 will not be as fast as L1 but can still have reasonable access time from the core perspective. The L3 cache as last-level cache (LLC) is also another auxiliary desk. However, the last level cache is frequently shared with other cores. The decision to keep the books accessed more frequently in the closest desk can be considered temporal locality, and the decision to put the books with a close subject on the closest desk to access soon is considered spatial locality.

1.3.1 Basic terminology

The following items present a more detailed definition of the concepts related to caches.

Cache miss/hit: When a read or write access to data that is not available in cache occurs, this is called a *cache miss*. Conversely, when the data is already in the cache, this is called a *cache hit*. The *miss rate* (the ratio between the number of misses and the number of accesses) is an interesting metric to evaluate the efficiency of a cache. Reducing cache misses is important to reduce the number of accesses to other levels of memory and contributes significantly to improve the execution time of an application and save energy. Finding the lowest miss rate for a set of benchmarks is the main quest of a cache designer.

Temporal locality: Temporal locality is the locality tendency in time, which stands that if a data is referenced now, it will tend to be referenced again soon.

Spatial Locality: Spatial Locality is the locality tendency in space, which stands that if a data is referenced now, the data close to it in memory tends to be referenced soon. Thus, this data is not brought in the cache alone, it comes within a block.

Cache block (or line): The cache block is the minimum addressable unit of data in a cache. Data transfer in or out of the cache is done in cache block unit, also referred sometimes as *cache line*. A common block size in current architectures is 64 or 128 bytes. The size of the cache is then defined by the block size and the number of blocks. Determining the good size of blocks and number of lines in the cache has been the subject of several studies. A bigger block increases the chance of spatial locality but also the risk of false sharing. Increasing the number of lines introduces hardware complexity that affects the performance. This kind of study (though interesting) is out of the scope of this thesis. In our work, we did not explore this dimension.

Private or shared cache: A private cache can only be accessed by its corresponding core. Since there is no interconnect, it is accessed with high bandwidth and low latency, e.g. L1 cache in Figure 1.1. A shared cache is accessed by all the cores. It has a constrained bandwidth to share data and due to interconnect there is longer access latency, e.g. L3 cache in Figure 1.1.

Inclusive or exclusive cache: In the inclusive cache, the lower-level cache includes all the data blocks of the higher-level cache. In exclusive cache, each data block can exist only in one level of cache hierarchy [67].

Replacement policy: Since the cache size is limited, the issue comes to evict one of the cache blocks to make room for the newly fetched block. There are many possible policies, which have also been intensively studied, to decide which cache block should be evicted from the cache. For example, the Least Recently Used (LRU) or the Least Frequently Used (LFU) cache block can be evicted. This policy is implemented in hardware, with its area and performance features.

Cache Associativity: Complementary to the replacement policy, the placement of the block in the locations of the cache can follow different schemes. In a *direct-mapped* scheme, each block has only one possible location in cache. Conversely, in a *fully-associative* scheme, a block can be located in any place. The middle range of schemes between *direct-mapped* and *fully-associative* is called *n-way set-associative*. In an n-way set-associative cache, a given block can be cached in any of n distinct locations. For instance, in a *two-way set-associative* cache, each block maps to a unique set but can be placed in two different locations. A direct-mapped cache is simple to implement but leads to a high miss rate. A

fully associative cache leads to the lowest miss rate, but at a high hardware cost. The set-associate cache stands as the good trade-off between direct-mapped and fully-associative, the challenge becoming to find the good trade-off.

A concrete example of an existing SMP is the Intel Xeon X5660 microprocessor, where the associativity configuration of the caches is as follows:

- L1 I-cache 4-way set associative
- L1 D-cache 8-way set associative
- L2 8-way set associative
- L3 16-way set associative

Write policy: When the data is written into a cache block present in the cache (write hit), it must be updated into the main memory. There are two main policies [70]:

- Write-through, a write to the cache is passed down to the main memory on the lower levels.
- Write-back, writes to the main memory are postponed: cache blocks which have been written into the cache are marked as dirty, and dirty cache blocks are written to the main memory when they are evicted from the cache (following the replacement policy).

1.3.2 Coherency

When a cache retrieves a data, it gets a *copy* of the value. The original value is still in the main memory. When the processor changes the value of the data in the cache, the two values for the same data are different. In the context of a single core, the core always sees the latest value as it is in the cache. When the cache block is evicted, the value is updated in the main memory. If it is retrieves back later by the processor, the core gets the up-to-date value. In the context of multiple cores sharing the same memory, this mechanism does not hold anymore. The data in a high level cache on one core might be different if this data has been modified by another thread on another core. There is thus a problem of *coherency*, different values of the same data at different locations.

Cache coherence protocols are designed to ensure the memory system to be coherent. Data coherency means that any read of data in cache returns the most recently written value of that data. Data coherency assures that values written by one processor are read by other processors. However, coherence says nothing about when writes will become visible. Moreover, coherence guarantees that “write”s to a particular location will be seen in order.

Complementary to coherency, consistency ensures that writes to different locations will be seen in an order that makes sense, given the source code.

The cache-coherence protocol needs to develop significant work to ensure that the data observed by different cores are not different, making two cores behave differently as they should behave identically by reading the same data.

Cache coherence protocol allows the processors to quickly access the replicated shared data located in their private cache while maintaining the coherency of shared data when the other processors modify its value. The standard method is to invalidate the local copy of data when it is updated by others [137].

For example, let's assume that $core_i$ and $core_j$ have a shared data in cache. When $core_i$ writes this data, it can be read by $core_j$ and vice versa. The cache coherence protocol handles the coherency of the data. There exists different coherence protocols. We present **MESI** [107, 8] which is one of the most well-known protocols used in current systems. The name MESI stands for four different states of the cache blocks (lines): Modified, Exclusive, Shared, and Invalid. These states are now described:

- **Modified:** The cache block is available only in the L1 cache of the current core that has modified it, but no other cache has the correct value of this block and it is dirty, i.e., the corresponding block in the main memory does not hold the correct value.
- **Exclusive:** The cache block is available only in the L1 cache of the current core, and it is clean, i.e., the corresponding block in the main memory does hold the same value.
- **Shared:** The cache block is available in the L1 cache of the current core, and it is clean in all of the caches in which it may be available.
- **Invalid:** The cache block is not available in the L1 cache of the current core.

In the mentioned example, in which $core_i$ and $core_j$ have shared cache block in cache and memory whose state is *shared* and clean, when $core_i$ wants to write (modify) this block, the cache coherence protocol changes the state of the block in the L1 cache of the writer core to *Modified* and the state of the block for other L1 caches which hold this block, including $core_j$ L1 changes to *Invalid*. Meanwhile, if the $core_j$ reads this block, a miss will occur and trigger the cache coherency to update this block with the new value. This updating operation usually takes up to 100 cycles [141].

Caches and their coherence protocols alone cannot do all the work to keep the data coherent. Support from the CPU at hardware as well as OS and libraries at the software

level is required to transmit the coherent cache features to the user level.

1.3.3 Synchronisation and caches

As already mentioned in the introduction, the use of caches exacerbates the synchronization overhead, as the coherency protocol is involved, imposing performance overhead that could cost more than one hundred cycles [141]. The tricky interactions between the atomic operations and the coherence state of the cache line are studied in [124], in 2020, which is surprisingly recent given the decades of history of both cache and synchronisation. The authors performed some experiments on various types of architectures for Compare-and-Swap and Fetch-and-Add operations, and show the impact of complex multi-level caches and memory hierarchy on the execution time of these operations. The paper also considers multi-socket architectures that are out of the scope of this thesis. Additionally, the results show that using atomics prevent any other operations, even if there is no dependencies, removing thus the instruction level parallelism that could be used, leading to 30× of bandwidth limitations.

The relationship between the core and the cache regarding AMO is specific to each architecture, but in order to keep compatibility with existing parallel applications relying on standard APIs, the software and hardware implementations should respect the contract. The responsibilities are shared between the hardware that must guarantee the atomic memory operations, and the low-level software libraries that rely on these operations.

As a summary, hardware solutions, including ISA-based synchronization, suggest to bypass the cache to avoid coherence overhead and propose dedicated techniques. These solutions suffer from classical pitfalls of hardware-implemented synchronization techniques, such as limited capacity (number of possible synchronization handled) and complexity. Regarding capacity, a hardware solution can propose an overflow management technique to switch to a software solution. However, software solutions relying on atomic operations in the context of SMP with caches suffer from several limitations (latency, bandwidth, serialization of operation) that prevent scalability of performance even though the hardware provides more cores and more memory.

1.4 Multi-core Simulators

A simulation-based methodology is followed in this thesis to functionally validate the hardware proposal at system level, on a SMP of several cores. This section introduces the basic concepts linked to simulation tools, sets our requirements, and reviews the different available tools. The section ends with a focus on Sniper, the simulation tool chosen for our experiments.

Both academy and industry require multi-core simulators to validate new techniques at software and hardware levels. The hardware complexity of multi-core has a significant impact on the simulation speed, which motivates the development of a dedicated field of research to propose different simulators, basically trading off between simulation speed and accuracy. Based on prior studies, multi-core simulators can be classified according to different metrics [43, 5]:

Simulation Level

Functional simulator: Functional simulators only simulate functionality of instruction set architecture (ISA) of target architecture and do not provide any timings estimate of the modeled architecture. Consequently functional simulators are the fastest simulators. Simple scalar [14], Pin tools [115], and Simics [89] are some example of this class of simulators.

Timing simulator: Timing simulators simulate the microarchitecture of processors and provide detailed statistics about the performance of the target system. Based on the details of the statistics, timing simulators are further divided into two categories:

- **Cycle-accurate simulators:** They model the microarchitecture of the target system at the cycle level. Consequently, due to the high level of statistics needed for this level of simulation, cycle-accurate simulators are relatively slow compared to other types of simulators. M-Sim [126] is an example of this category of simulators.
- **Event-driven and interval simulators:** Such simulators break the simulation into different events like cache-misses, number of instructions, etc. SESC [117] and RSIM [63] are two examples of these types of simulators.

Integrated Functional and Timing simulator: Functional and Timing simulators can be coupled together to achieve better simulation's speed/accuracy. Sniper [31], Gem5 [22],

and ZSim [119] are few examples of this class of simulators.

Scope of Simulation

Full-system simulators: Full-system simulators simulate the entire system so that complete software stacks (application, OS, etc.) can run on the simulator. Due to the detailed statistics of these simulators, they are usually slower in comparison to other categories. Gem5 [22], MARSSx86 [13], and PTLsim [147] are three examples of full system simulators.

User-level simulators: User-level simulators do not simulate the full software stacks and only focus on the target workload execution. ZSim [119], Sniper [31], and Graphite [97] are widely used available user-level simulators.

Simulation Inputs

Trace-driven: Trace-driven simulators use instruction and address traces of application as an input and provide detailed microarchitectural timing simulation for the target platform. Sniper [31], Graphite [97] are two examples of trace-driven simulators.

Execution-driven: In this approach, simulators directly provide timing simulation from output of functional simulations and hence eliminate the need for trace storage. ZSim [119], RSIM [63] and PTLSim [147] are some examples of execution-driven simulators.

Thesis's simulation requirements

The research developed in this thesis has some non-functional requirements regarding the simulation properties previously presented. The following paragraphs detail such requirements:

Simulation speed: We expect our simulators to simulate at least multiple seconds of a parallel benchmark on modern multiprocessors below 24 hours time frame. This limitation is due to the large number of experiments we need to run in our study.

Table 1.2 – Overview of recent CPU simulators and their main characteristics

Name	Last update Release year	Full-system/ User-level	Trace-/Execution-driven	Multi-threaded/ Sequential	CPU complexity
MARSSx86	2011	Full-system	Execution	Sequential	OoO
gem5	Recently	Both	Both	Sequential	OoO
ZSim	2018	User-level	Execution	Multi-threaded	OoO
Sniper	Recently	User-level	Both	Multi-threaded	OoO

Simulation scope: We are interested in multiprocessor simulations. Our simulators should either be full-system or user-level simulators that can emulate synchronization between different threads with high accuracy.

Simulation target hardware: Our study requires to focus on general purpose multiprocessor architecture. The capability of the simulator to simulate a low-power multiprocessor would be interesting, since the applications studied in this thesis are image and video oriented, typical of the embedded domain.

Simulation error: Our simulator should be validated against actual hardware for parallel applications for a high number of cores. This is an important requirement in our study. Moreover, we should be aware of sources of error of the simulator and hence, make sure these sources do not impact the results and conclusions of simulations.

Simulators publicity: Our simulators must be widely used and accepted by the computer architecture community. Moreover, strong user support is a plus.

Available simulators

Table 1.2 lists recent publicly available timing simulators that are widely used by computer architecture community [5].

MARSSx86 [13] is first publicly available x86 full-system simulator released in 2011. MARSSx86 is based on two open-source simulators, QEMU [18] and PTLsim [147]. It is built on top of QEMU [18] to simulate OS and for simulating processor, MARSSx86 expands PTLsim [147] which is a cycle-accurate simulator for generic processors. MARSSx86 has been initially validated against actual hardware using PARSEC [21] parallel benchmarks with average error rate of 11%. Akram et al. update and evaluate MARSSx86 w.r.t.

x86 Haswell hardware using SPEC benchmarks and report 27% error on average [5]. Currently, MARSSx86 does not benefit from strong community support. Moreover, its ISA implementation is outdated and cannot provide accurate timing support for many x86 instructions.

Gem5 [22] is built by integration of processor pipeline M5 [23] simulator and GEMS [93] memory hierarchy simulator. Gem5 is a cycle accurate simulator capable of simulating heterogeneous multiprocessor at both full-system level and user-level with strong community support. Due to its detailed sequential implementation, despite a very strong ability to simulate various systems, gem5 is a slow simulator relative to other alternatives.

Butko et al. updated gem5 and validated it w.r.t. actual ARM A9 hardware using SPLASH-2 parallel benchmarks with average error rate of 6% [30]. Later, Gutierrez et al. updated gem5 and evaluate it against ARM A15 hardware using PARSEC parallel benchmarks with average error rate of 16% [56]. There are not many successful attempts in evaluation of gem5 simulator against recent x86 hardware. Akram et al. update and evaluate gem5 w.r.t. x86 Haswell hardware using SPEC benchmarks and report 40% error rate on average [5].

ZSim [119] is a fast, execution-driven, user-level simulator, developed at MIT, to simulate large scale systems. ZSim is originally developed to simulate ZCache [118] as a new cache architecture for multicore systems. Functional simulation of ZSim is handled by Pin tools. ZSim originally evaluated against PARSEC parallel benchmarks with average error rate of 11%. Later Sanchez et al. update and evaluate ZSim against SandyBridge architecture and report 28% simulation error on average [140]. Akram et al. update and evaluate ZSim w.r.t. x86 Haswell hardware using SPEC benchmarks and report 27% error on average. Currently, ZSim benefits from community support. The main challenge with ZSim is in its implementation. ZSim has two phases of simulation known as Bound and Weave. In Bound phase ZSim simulates each core isolated and in Weave phase, ZSim resolves the conflicts over shared resources. This complex timing model provides very fast simulation speed at the cost of inaccuracy in simulating memory system and false sharing effects [83, 140].

Sniper [31] is an execution-driven/trace-driven, user-level, parallel simulator for simulating large scale multicore systems based on interval simulation [49]. Sniper is built on top of Graphite [97] simulator, which supports various one-IPC core models. To perform functional simulation, Sniper incorporates Pin [115] dynamic instrumentation framework.

Although Sniper is a user-level simulator, it can simulate synchronization and locking mechanism with great accuracy. Sniper has been initially validated against actual hardware (Intel Xeon) using SPLASH-2 benchmark with an average error of 25% for multicore simulation [31]. Later, Sniper is updated and validated with an error rate of 11.1% using SPLASH-2 benchmark [33]. Recent studies updated and evaluated gem5, Sniper, ZSim, and MARSSx86 simulators against actual Haswell processors. Sniper exhibits short simulation time relative to other simulators (7x times faster simulation speed relative to cycle-accurate gem5 for SPEC benchmark) while maintaining high accuracy [6, 5]. Recently, a new version of Sniper was released with the support of RISC-V and ARM ISA [90, 4].

Sniper

We use the Sniper simulator since it has been compared to other simulators and is also validated against real hardware [31, 33, 6, 5]. We use Sniper for both runtime and energy/power evaluations as it has also been cited by more than 950 works according to Google Scholar. Sniper integrates McPAT [82] that is a widely used microprocessor power and energy model with more that 2,800 citations to date according Google Scholar. Sniper meets the demands of this work which are accurate core, cache, and memory system for parallel benchmarks as Sniper's core, cache, and memory system are validated against actual hardware for parallel benchmarks in multiple studies [31, 33, 6, 5]. Other simulators are not used because:

- gem5 is a very slow simulator relative to other x86 alternatives and does not provide significant additional accuracy when compared to other simulators for our benchmarks [5, 11].
- MARSSx86 is outdated and does not have any community support anymore.
- ZSim has a very complex timing model (Bound, Weave) that prevents accurate modeling of false sharing [136] and other memory system modifications [140, 83, 119].

Sniper Accuracy Three simulation models are integrated in Sniper [31]: one-IPC, interval, and instruction-window-centric (IW-centric). The **one-IPC** model [31] assumes an in-order single-issue at a rate of one instruction per cycle; hence it is named as one-IPC or 'one instruction per cycle. The one-IPC model assumes that the branch prediction is perfect, so it does not simulate the branch predictor. However, it simulates the multiple

levels of the cache hierarchy. The cache model assumes that the processor model does not incur any penalty (latency) for load and store of L1 data cache hit and has an execution latency of one cycle. For all other caches, it considers the miss penalties. In particular, an L1 instruction cache miss incurs a penalty of L2 cache data access latency; an L2 cache miss incurs a penalty of L3 cache data access latency or main memory access time in the absence of an L3 cache. **Interval** simulation [49] is a recent simulation approach for simulating multi-core and multiprocessor system at a higher level of abstraction compared to the detailed cycle-accurate simulation. This model is based on miss events (branch mispredictions, cache and TLB misses), i.e, it divides execution time of the instruction stream through the pipeline into intervals separated by disruptive miss events. The total execution time is then determined by aggregating the execution time over all intervals. Branch predictor, memory hierarchy, cache coherence and interconnection network simulators determine the miss events; the analytical model derives the timing for each interval. Instruction-window centric (**IW-centric**) simulation [33] is a new high-level core model that combines the idea of interval modeling with a detailed simulation model of the instruction window, or reorder buffer (ROB). However, the interval simulation methodology calculates the instruction-level parallelism (ILP) of an application analytically, based on the intervals, IW-centric simulation models micro operation dependencies and issue timing in detail. This model provides more accuracy considering fine-grained events in cost of a small reduction in simulation speed. In terms of modeling accuracy, the IW-centric model can be used as a step closer to cycle-level simulation.

One-IPC model is the fastest simulation model with the most abstract core model, and hence highest inaccuracy. Carlson et al. first present interval-based simulation of Sniper with an average error rate of 25% compared to actual hardware for SPLASH-2 parallel benchmarks [31]. Later Carlson et al. proposed a new core model (a.k.a. IW-centric) to bridge the gap between interval-based simulation and cycle-accurate simulation. They show the IW-centric core model is more accurate with only 11.1% error rate w.r.t. actual hardware but at the cost of slower simulation speed [33].

To perform our experiments, we must choose one of the simulation models provided by Sniper. Additionally, the chosen model must cover our research goal in terms of accuracy and reasonable simulation time. Since the aim of our research is performance evaluation, we run one of our application benchmarks with these three simulations models and also on real Intel corei5 machine to compare the accuracy of these simulation models. Figure 1.2 shows the results of our evaluation experiments. The X-axis shows the number of cores we

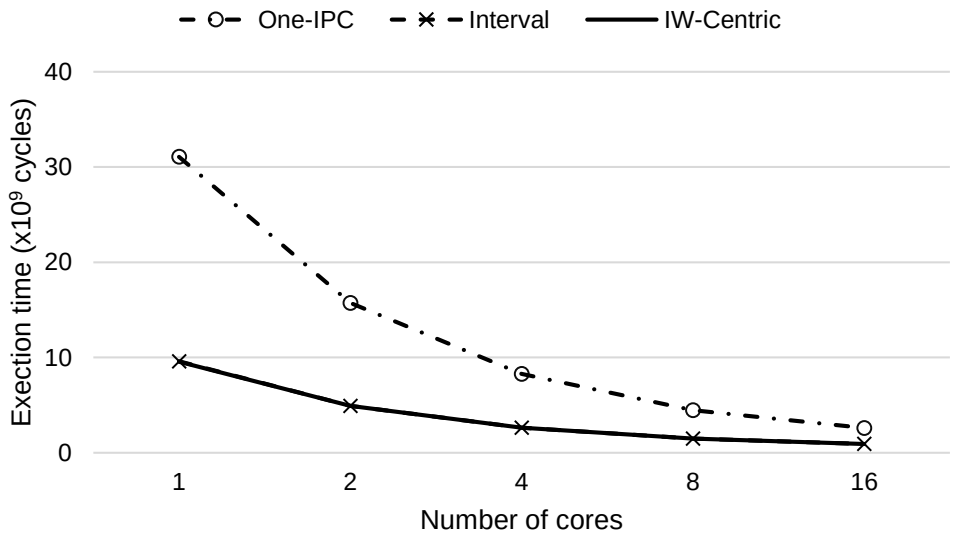


Figure 1.2 – simulator evaluation

used during the experiments, and the Y-axis represents Sniper's execution time results in billion cycles. As it shown in the figure, one-IPC model compared to the real architecture shows the most inaccuracy. We witness that despite the inaccurate one-IPC model, the interval model presents the same result of execution time as the accurate model of IW-centric. Regards to simulation speed, as also presented by Carlson et al. [33], IW-centric model is 1.5× slowdown compared to interval model simulation. We decided to continue our research experiments with the interval model, which is a much faster model than IW-centric and yet delivers a similar simulation result.

1.5 Summary and Concluding Comments

By providing high performance and general-purpose programmability, SMPs, among other multi-core processors, have been popular. Additionally, SMPs, by maintaining the image of global shared address space as a single memory, facilitate the programming of ever increasing variety, complexity, and performance requirements of parallel applications. SMPs use a cache-coherent managed memory hierarchy which requires data coherency consistency. Increasing the number of parallel core processors in these architectures can provide more performance while maintaining the power characteristics. Parallel processes running on SMP require synchronization, and also the synchronization primitives need extra coherency and consistency maintenance.

Synchronization is a well-explored field and as discussed a variety of techniques from hardware and ISA support to purely software solutions have been proposed. Almost all of these techniques are developed for generic settings that do not consider the memory behavior in the dataflow applications. These generalized approaches create a bottleneck for dataflow applications since they require a large number of producer-consumer threads to be synchronized. As such, the centralized nature of these synchronization mechanisms serialized the execution of dataflow applications. A review of the state-of-the art techniques for synchronization shows that only one approach, already developed in the team, makes use of the specific synchronization needs of dataflow applications, that are explicitly available through the firing rules. The goal of this thesis is to go further. In this thesis, we propose such a solution through a specialized hardware unit that augments the processor core to accelerate synchronization and offloads its burden from the core itself.

Chapter 2

Dataflow Model of Computation

The dataflow Model of Computation (MoC) has a long history of several decades of research. The history of dataflow can be traced back to the time that Petri invented "Petri nets" in 1962 [28] and Estrin and Turn proposed an early dataflow model in 1963 [45]. In 1966, Karp and Miller studied computation graphs without branches or merges and Rodriguez extended and formalized Estrin's model in 1969 [91]. Chamberlain proposed a single assignment language for dataflow in 1971 [35] and Kahn studied a simple parallel process language with queues as edges in 1974 [71]. On the hardware side, we can give credit to the dataflow architecture designed by Dennis [37], where edges were one token buffers. In 1977, Arvind and Gostelow [12], and separately Gurd and Watson [55] proposed a "tagged token" dataflow model, where the edges behave like bags.

This chapter presents some generalities about dataflow MoC, and the two main categories of dataflow MoC: static and dynamic. This chapter presents in more details one MoC for each category, chosen for our experiments: SDF for static MoC, and PiSDF for *parametric* MoC, a subclass of dynamic MoCs [26]. The interested reader can refer to existing surveys for a comprehensive overview of either static MoCs [57] or dynamic MoCs [20]. The chapter then presents the framework used for our experiments. The chapter ends with a review of studies related to the impact of cache configurations in the execution of parallel applications.

2.1 Generalities

An application specified in a dataflow MoC is usually represented as a graph of processing elements connected through data dependencies, which is called a dataflow graph. This kind of graph can be specified through a textual form, like many other programming languages, but a graphical view is a convenient way to represent dataflow applications. Such a graphical view is used by the tools chosen for this thesis and is used also throughout this document.

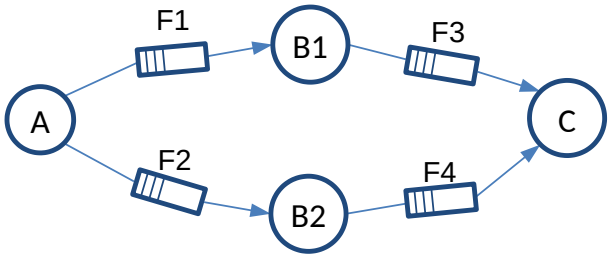


Figure 2.1 – Dataflow graph

A dataflow graph is depicted in Figure 2.1. In this graph, vertices ($A1$, $B1$, $B2$ and $C1$) are processing elements called actors which exchange data tokens through the unidirectional edges, which are data FIFOs (F_1, F_2, F_3 and F_4). The actors consume/read the data tokens from their input FIFOs, process them and produce/write new tokens into the output FIFOs. An actor is a *non-preemptive* process that can initiate the execution (fire) only if the data is available in its input FIFOs. These conditions are called firing rules. The formal model defines rules on the input only, and considers unbounded FIFOs. However, when it comes to the implementation on a hardware platform, the FIFOs are bounded and the space available in the output FIFOs also becomes one condition for firing.

The ability of dataflow graphs to describe various applications with different features can be evaluated by different properties, including: Expressiveness, Predictability, Efficiency, and Analyzability [38]. Figure 2.2 illustrates these properties.

Expressiveness: The Expressiveness property is the ability of the MoC to specify the complexity of the application behavior. A highly expressive language gives the ability to describe any kind of application. Conversely, a low expressiveness restricts to use a limit set of computational behaviors. A MoC that gives the theoretical expressive power to model any computation is Turing-complete.

Practicality: Practicality is the ability given to the developers to describe "in practice" the functional behavior of an application. It is related to the *ease of use* through a given language that can be concise, intuitive, and easy to read.

Efficiency: Efficiency represents the performance of the application when it's running on a hardware platform [134].

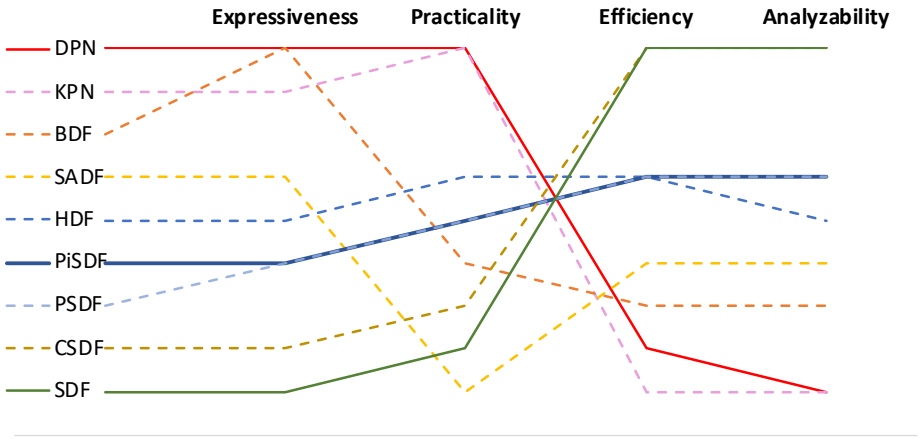


Figure 2.2 – Comparison of different dataflow MoCs [148]

Analyzability: Analyzability is the ability to analyze and characterize the application at compile time.

The different MoCs presented in Figure 2.2 are not described individually, but it shows that it is difficult to concentrate into a single MoC all these properties. A MoC with a high expressiveness offers a low efficiency and analyzability. Conversely, restricting the possibilities to model any kind of computation gives the benefit of a better analyzability and efficiency. This chapter follows the classical classification of dataflow MoCs in two main categories: static dataflow MoCs and dynamic dataflow MoCs [26].

2.2 Static Dataflow MoCs

The family of static dataflow MoCs can be characterized by the common feature of a constant rate in the consumption and production of tokens, which offers interesting properties like determinism, decidability, and compile-time optimizations.

Among the different static MoCs, Synchronous DataFlow (SDF) is the most basic dataflow model to describe an application. As shown in Figure 2.2, this model presents the best efficiency and ability to characterize the application at compile time (Analyzability); however, this simple model has the lowest Expressiveness and Practicality. The Synchronous term relates to the fixed value of the rate of generation and consumption of tokens, i.e., all the actors produce/consume a fixed number of tokens in the whole graph. This model has significant advantages. First, high analyzability, which means it can be analyzed mathematically. Second, the determinism of firing rules provides the opportunity of

having optimized mapping and scheduling at compile time for the applications. However, SDF semantics are not able to describe a data-dependent application. In data-dependent applications, the firing of the actors may depend on the input data at run-time [123].

In **homogeneous** SDF (a.k.a **Single-rate** SDF), an actor starts firing when there is one data token in its input ports; also, it produces one data token in its output ports. Therefore, in this scenario, the scheduler must ensure that the producer actor fires (executes) before the consumer actor, and also each iteration of the model consists of only one firing of each actor. In general, the actors produce/consume more than one single data token, i.e., they require multiple data tokens as input and produce multiple tokens as output. Therefore, the SDF scheduler can support more complicated SDF models than homogeneous such as **multirate** SDF model. In multirate SDF, the actors' production/-consumption data rate is not identical [113]. Indeed for multirate SDF, the scheduler transformed the graph to a middle single rate graph (Directed Acyclic Graph (DAG)).

Various dataflow MoCs extensions are introduced to support more practical applications while preserving the predictability and analyzability of the model to a feasible extent. These models are to some extend more general than SDFs, and are not considered in this thesis. Though the contribution presented in this thesis is not dedicated to SDF only, some more work is needed to study how to adapt our proposal to other static MoCs.

The graph of a synthetic application modeled by SDF is depicted in Figure 2.3. A synthetic application is a program that is constructed to try to match the characteristics of a large set of programs [143]. We use this application as a representative example to discuss various aspects of the work. We refer to this application as the synthetic application. This application includes three main actors; *Producer*, *Processing* and *Consumer*. The *Producer* actor generates and writes data tokens. Then the *Processing* actor processes the data tokens. At last all the processed data tokens are consumed by the *Consumer* actor. In the figure, the blue blocks are constant parameters which are known at compile time. The *msg_size* determines the size of the output and input FIFO of the *Producer* and *Consumer* respectively. In this graph the *Processing* actor consumption/production rate of the tokens is determined indirectly by *slice_size* parameter through the following equation:

$$Token\ rate = \frac{msg\_sizes}{slice\_size} \tag{2.1}$$

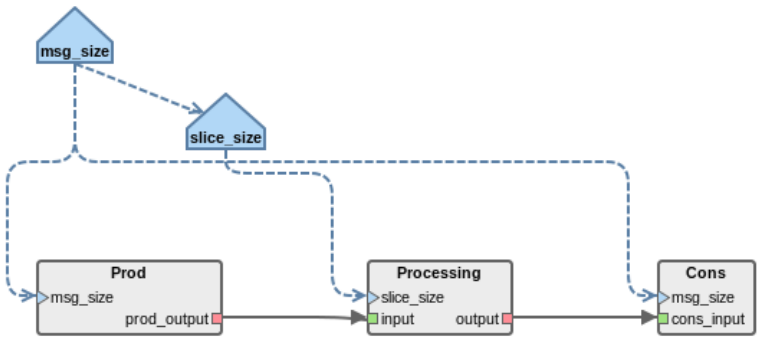


Figure 2.3 – SDF Dataflow graph of simple producer-consumer application

2.3 Dynamic Dataflow MoCs

2.3.1 Generalities

The second category of dataflow MoCs is dynamic dataflow MoC, which achieves a more powerful and comprehensive model. Dynamic dataflow models encompass modeling techniques in which the consumption and production rates of actors may vary. It is thus not possible to define at compile-time a complete order of the actors. Runtime techniques are therefore needed to check the firing rules. As shown in Figure 2.2, the DPN (Data Process Network) MoC [79] is the most comprehensive and generalized MoC with the best ability to express the behavior of complex applications. Indeed, DPN has the highest Expressiveness, equivalent to Turing machine [29] which means it can describe any application.

2.3.2 PiSDF: a Reconfigurable Dataflow Model

This section relates only a subset of dynamic models, which are reconfigurable dataflow models [26], through the presentation of PiSDF that is used for our experiments. Parameterized Interfaced Synchronous Dataflow graphs (PiSDF) can be viewed as an extended SDF model, and solves non-reconfigurability characteristic of the SDF. In PiSDF, in addition to the normal actors, there are configurable actors which are in charge of generating parameters which determine the rate of consumption and production of data tokens (firing rules). These configurable actors can be executed in specific points of execution time of the application and reconfigure the firing rules at run time [102]. Applications modeled by PiSDF graphs are classified as reconfigurable dataflow applications although reconfigurable dataflow models are not limited to PiSDFs. PiSDF graph of the

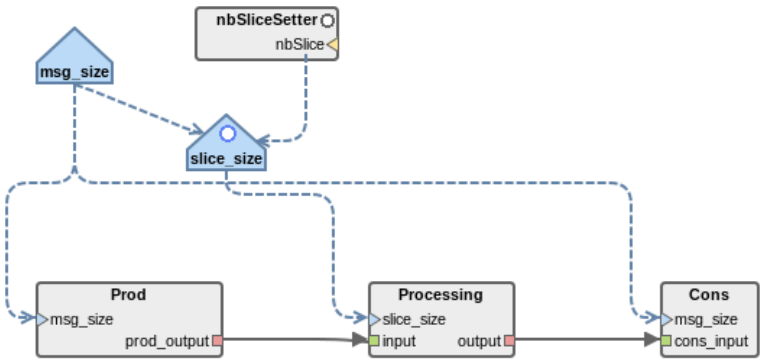


Figure 2.4 – PiSDF Dataflow graph of Synthetic producer-consumer application

synthetic application is represented in Figure 2.4. In this graph, the reconfigurable actor, *nbSliceSetter*, is responsible for configuring the *slice_size* which sets the *Processing* actor consumption/production rate of the token by the Equation 2.1. In other words this actor reconfigures the token rate of the graph.

2.4 Dataflow Prototyping Frameworks

2.4.1 PREESM

Several frameworks for dataflow modeling exist. The goal of this section is not to exhaustively present all of them but to focus on the one chosen for our experiments. In this thesis, we use PREESM [110], a state-of-the-art open-source rapid prototyping tool, to generate code of the dataflow applications as benchmark of our study. PREESM is a software framework developed at the Institute of Electronics and Telecommunications of Rennes (IETR) in 2007. It provides a Graphical User Interface (GUI) based on Eclipse for the designer to generate a deadlock-free C code for dataflow applications modeled with dataflow graphs for heterogeneous multi/many-core embedded systems.

The framework allows the developer to model applications following a static or reconfigurable approach. In static approach, the application is generated already with the optimal mapping and scheduling since data token production rate is deterministic. For reconfigurable applications, a Run-time Management Layer (RML) is responsible for dynamically mapping and scheduling the actors according to workload characteristics. This procedure will be discussed in detailed in Section 2.4.2.

Figure 2.5 depicts the infrastructure of PREESM for generating code for dataflow

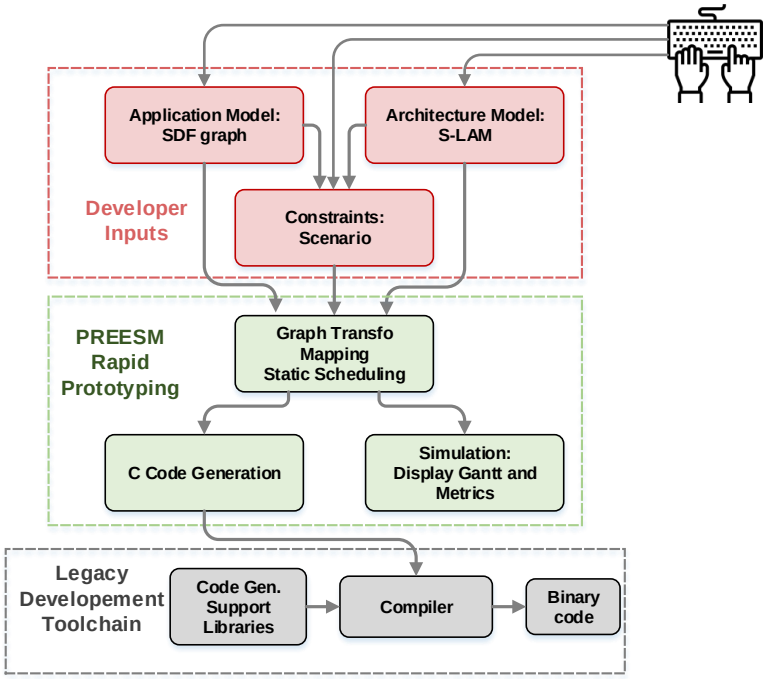


Figure 2.5 – Detailed overview of PREESM framework.

applications from a SDF or PiSDF model. As illustrated in the figure, the flow of PREESM includes three principle steps: *Developer Inputs*, *PREESM Rapid Prototyping workflow*, and *Legacy Development Toolchain*. At the first step, *Developer Inputs*, the developer provides the input information for PREESM framework. The information includes:

- **Application Model:** In this input, the developer provides the algorithm of dataflow application specified in SDF or PiSDF.
- **Architecture Model:** This input is the System-Level Architecture Model (S-LAM) that provides a high-level architecture description of the target multi-core system for which the application is going to be generated.
- **Constraints:** The developer, specifies the design constraints [40] (e.g., latency, throughput, load balancing, etc.) related to scheduling, mapping of the actors on the cores and their timing on each type of processing cores.

At a second step, PREESM generates a pthread C code of dataflow application for a target architecture, considering the application model, architecture model, and related constraints.

- **Mapping and Scheduling:** As discussed earlier, SDF graph has more complicated extensions such as multirate SDF and IBSDF. To prepare the scheduling

and mapping of the input SDF graph, PREESM transforms it to the basic SDF graph by applying hierarchy flattening and transforming the graph to the Single-rate Directed Acyclic Graph (srDAG). Thereafter, it decides optimized mapping and scheduling out of throughput, latency and load balancing for the transformed graph. Moreover, PREESM also applies memory management methods to map the actors [40].

- **Visual output:** PREESM provides the Gantt diagram of the application with memory and load balancing metrics of mapping for developer.
- **C Code Generation:** The framework generates a parallel C code for the target multi-core architecture that relies on pthread library. PREESM bundles all the actors mapped on each core in one thread, such that there is one thread per core. Additionally, the code relies on a semaphore based library to synchronize the threads.

At the last step, PREESM supplies runtime libraries to support the execution of generated application code on various supported architectures. These libraries abstract the communication and synchronization mechanisms of the target architecture and support calls to high-level primitives in the generated code. As mapping and scheduling decisions are statically made during the PREESM workflow execution, the execution of the application is self-timed [130]. In a self-timed execution, the execution time of an actor is not pre-determined and the order of actor firings is guaranteed by inter-core synchronizations and communications.

2.4.2 SPiDER: Runtime Management Layer for Dataflow Applications

As previously discussed, SDF graphs can only model the dataflow application with non-reconfigurable behavior and provide the opportunity of optimized mapping and scheduling at compile time though they have limitations in expressiveness. Therefore, for the applications whose behavior change based on the input data at run time (reconfigurable), the Parametric Interfaced Based SDF (PiSDF) model can be used.

To execute the code generated from a reconfigurable dataflow application, PREESM utilizes a Runtime Management Layer (RML) called Synchronous Parameterized and Interfaced Dataflow Embedded Runtime (SPiDER) [62]. For the reconfigurable dataflow applications code, PREESM generates an interface that becomes part of the generated

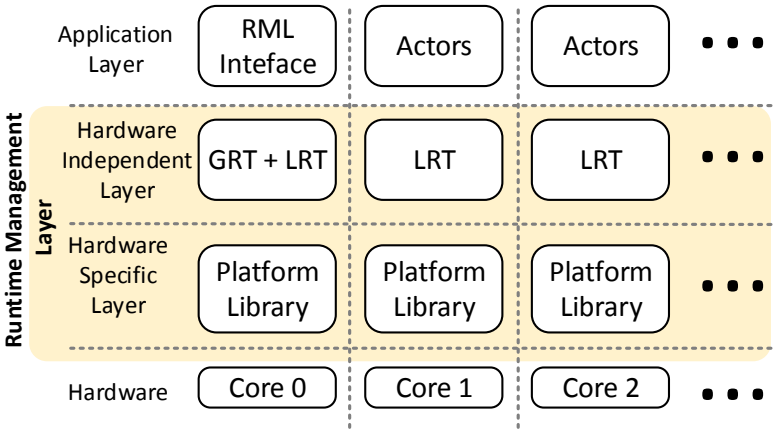


Figure 2.6 – SPiDER infrastructure as a Runtime Management Layer.

code of the application and makes the bridge between the application and the RML. This interface has two purposes: (*i*) to provide to RML the constraints of application and properties of the target platform, (*ii*) to provide to RML the functions of the actors to start them according to the runtime scheduling and mapping decisions. SPiDER RML manages reconfigurable applications through hierarchical runtime management composed of one global manager (GRT) and n local managers (LRTs) (where n is the number of threads running the application). GRT is responsible for managing the application graph, performing mapping, and scheduling. LRTs are responsible for executing actors by running the jobs provided by GRT. Figure 2.6 shows the SPiDER Runtime Management Layer structure. The first layer is the application layer, having the RML interface and the actors. The RML interface calls SPiDER RML and initializing it with architectural and application properties. At the bottom of the application layer is the RML layer, divided between Hardware Independent Layer (HIL) and Hardware Specific Layer (HSL). HIL implements the hierarchical management with one GRT and several LRTs. HSL implements architectural dependent tasks (as communication and synchronization) according to the target platform.

2.5 Cache Studies for Parallel Applications

We are interested in the specific case of dataflow application behavior in the presence of cache, which is insufficiently studied. Thus, the literature review was extended to all kinds of parallel applications running on an SMP. This section presents the related studies evaluating the performance of parallel applications in a multi-core system with cache

Table 2.1 – Related works studying the cache impact in parallel applications.

Author (et al.)	**Proposal**	**Contribution**	**Benchmark**
Garcia [48]	N.A. (Evaluation work)	Evaluation of impact of LLC sharing	Heterogeneous applications
Slingerland [127]	N.A. (Evaluation work)	Cache profile of multimedia applications	Multimedia applications
Alves [7]	N.A. (Evaluation work)	Evaluation of L2 properties	NAS parallel applications
Domagala [41]	Splitting nested loops	Increased Data locality	StreamIt
Maghazeh [88]	Splitting GPU kernels to sub-kernel and data input into L2 size	Increased Data locality + Decreased cache miss rate	GPU-based applications
Fraguela [17]	Strategy to improve cache usage in dataflow	Minimize communication among processes involved	Cholesky decomposition
This work	*Use of two dynamic memory manag. methods (CoW, NTM)*	*Cache configuration evaluation + Reduction in memory copy penalties*	*Static Dataflow applications*

memory hierarchy. While generating the parallel code from dataflow specification for an SMP, PREESM framework implicitly assumes a coherent shared memory. Besides, the features of the cache hierarchy are not taken into account. It is, therefore, possible that the same generated code behaves differently according to the cache features. The first part of this work is thus to study the impact of the cache features on dataflow applications. Table 2.1 summarizes the main characteristics of the related works addressing the cache impact in parallel applications. Several studies [48, 127, 86, 7] have provided an evaluation of the impact of the levels of caches on the performance of parallel applications, but they did not delve into dataflow applications.

Garcia et al. study the cache effect on the performance of heterogeneous applications. These applications comprise CPU and GPU code which is the reason they are considered as heterogeneous applications [48]. They have evaluated the impact of Last Level Cache (LLC) shared in GPU-CPU co-design platform for these applications. According to their study, applications with low data interaction between GPU and CPU are sped up slightly by sharing the LLC. Data sharing of LLC minimizes memory access time and dynamic power and accelerates synchronization for fine-grained synchronization applications.

The memory behavior for multimedia workloads in the presence of cache is evaluated by Slingerland and Smith [127]. They appraised the data miss rate of applications considering data cache size, associativity, and line size parameters. The authors observed that multimedia applications benefit from longer data cache lines and have more data than instruction miss rate in comparison to other workloads. The experiment results reveal that most of the multimedia applications need 32 KB data cache size to have less than 1% cache miss rate, while other types of applications (3D graphics, document processing) do not reach the same behavior.

Lotfi-Kamran et al. [86], during the study of the impact of LLC size on performance for Cloud workloads, reached this conclusion that sharing LLC with bigger size among more cores does not help the performance of the application. They show that increasing the cache size is not beneficial for most workloads as the decrease in miss rate is usually offset by increased access latency. The work of Alves et al. [7] investigates the impact of L2 sharing in order to find the best cache organization at this level. They use NAS Parallel Benchmark [15] with a heterogeneous workload set, and a 32-core SMP with two levels of cache. The work changes the sharing, size, associativity, and line size in the L2. Among the main results, it was observed a performance (speedup) decrease when more cores share the L2 cache, even when two cores share the same L2. Increasing line size (64 bytes to 128 bytes) contributed to -32% in cache misses and +1.95% in speedup. The work does not address 3 level caches nor dataflow applications.

Some other works propose solutions at the application or hardware level to aid parallel applications exploit the cache hierarchy features efficiently [41, 88]. In Domagala et al. [41], researchers extended the concept of tiling to the dataflow model to increase the data locality of applications for better performance by splitting iterations of nested loops. However, this type of optimization does not address the fine-grain inter-actor (i.e., inter-tasks) relation.

In Maghazeh et al. [88], a method is proposed for GPU-based applications by splitting both the GPU kernel into sub-kernels and input data into tiles in size of GPU L2 cache. Their work is intended to accelerate applications whose performance is bounded by memory latency. The method increases data locality, as the sub-kernels are scheduled to have the minor cache miss rate for GPU applications over various settings. However, the method requires source code modification and does not target the dataflow model.

The existing related work regarding the studies of the memory behavior specifically for dataflow applications is very limited. The work done by Fraguela et al. can be cited [17]. Fraguela et al. [17] propose the concept of a software cache with an auto-tuning method to configure its size according to each application. The approach is based on Unified Parallel C++ (UPC++) library. It consists of an algorithm called periodically, which dynamically re-allocates the software cache size. Results show that the software cache can reduce the communication among actors due to the efficient cache sizing and allocation, presenting a hit rate just 0.27% higher than an optimal scenario.

The main novelties of our work regarding the related works, including non-dataflow and dataflow studies, are twofold: (*i*) we evaluate a wide range of cache configurations in a

multi-core architecture, including existing configurations and configurations not available yet but corresponding to what might be available in the near future when technology scaling enables integrating ever bigger caches; and (*ii*) we use two existing memory management methods for dataflow applications, which can reduce the memory copies penalties in numbers and latency, leading to an improved application execution time and energy consumption. Regarding contribution (*i*), works [127, 48, 7] are also evaluation works. However, in [127] the benchmark is limited to multimedia applications, in [48] the focus is the interaction between the CPU and GPU, by addressing a heterogeneous CPU-GPU set of applications but not considering dataflow, and in [48] the authors did not consider a 3-level cache either the dataflow application profile. From this literature review, there is a lack of a comprehensive evaluation of the impact with 3-levels caches and targeting dataflow applications. The present work fills that void.

Regarding contribution (*ii*), the present work differs from related works [41, 88, 133, 17] by several aspects. Specifically, we are interested in: (*i*) keeping the original dataflow model granularity (differently from [41, 88]); (*ii*) not making modification in the Linux-based kernel, or any part of the OS (as required by [17]). We endorse that the memory management techniques we use are not new. The goal of using these memory management techniques is to replace the memory copy (memcpys) operations and to observe their impact on cache and on the overall performance of dataflow applications, a study that is lacking in the literature.

2.6 Summary and Concluding Comments

This chapter presented generalities about dataflow MoC, and detailed two MoCs, SDF that can be classified in static models category, and PiSDF that an be classified in dynamic model category. The dataflow MoCs allow the developer for designing a complex program with fine-grain parallelism. Dataflow MoCs define the applications by a graph of the processing functions that execute and exchange the data chunks as input/output data.

In addition, we have also presented a summary of the state-of-the-art studies related to our contributions. First, we reviewed the studies on cache characterization in various platforms for different benchmarks. Many works have evaluated cache impact on different parallel applications. However, no work evaluates the cache-based systems for dataflow applications.

Chapter 3

Cache evaluation and dynamic memory management techniques

This chapter presents the first contribution of this thesis: the study of dataflow applications running on a SMP according to different cache configurations, providing experimental results that demonstrate their impact on the application's execution time, system energy, and cache miss. It also includes the automatic use of dynamic memory management techniques to improve memory reuse. This chapter is also described in a conference paper accepted at DASIP [50] and extended in Journal of Signal Processing Systems (JSPS) [51].

Dataflow models can naturally use parallel resources employing actors that run in parallel while consuming and producing data tokens. Several tokens can be produced and consumed simultaneously, but a token is produced and consumed only once. This feature favors spatial data locality. While the cache hierarchy also exploits temporal locality, a dataflow program may benefit from the latter for instructions and spatial locality for data as consecutive tokens are usually involved. Therefore, dataflow applications' performance should be improved with the increasing size of caches. However, this study shows that such an assumption does not hold regarding multiple cache-based architecture designs.

Although the state-of-the-art techniques can lead to theoretical optimal schedules, this study demonstrates that even optimally scheduled applications (generated with the assistance of rapid prototyping tools) do not scale as desired with the increasing number of cores, cache levels, size and cache sharing factor. As expected, the memory contention is of utmost importance, and the CPU load-based actor mapping used in the experiments does not lead to the best execution time.

For the experiments, we consider several configurations, including non-available yet platforms or non-realistic cache configurations, and use the Sniper simulator [31] to foresee the scalability of the considered dataflow applications.

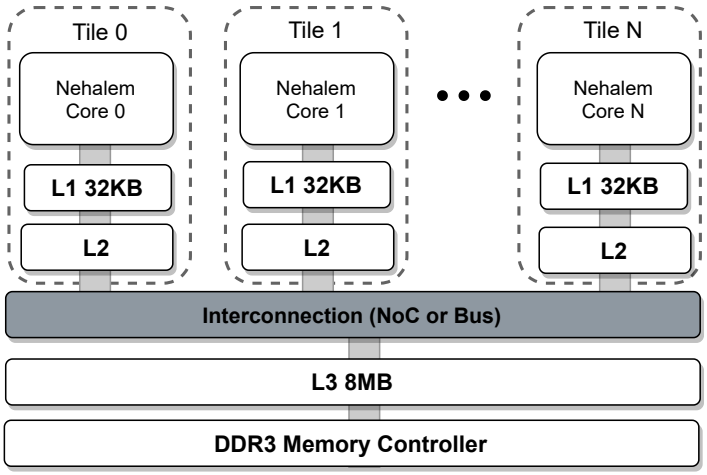


Figure 3.1 – Architecture overview of the baseline multi-core model.

3.1 Multi-Core Model Used in Experiments

This section presents the multi-core architecture model adopted in this study. Figure 3.1 presents the architecture overview. We focus on detailing the memory hierarchy since it is the target of this work. The architecture is based on the Intel Xeon X5500 chip. Each core implements the Nehalem Intel microarchitecture [76], having a private L1 cache with 32KB, a private L2 cache with 256KB, and a shared by four cores L3 cache with 8MB. The chip also includes a triple channel DRAM memory controller to interface with off-chip DRAM memories.

The interconnection is bus-based with a 20-bits width and provides 12.8 GB/s per link in each direction (25.6 GB/s total).

The architecture depicted in Figure 3.1 is the reference multi-core model. The actual goal is to exploit different core counts and cache configurations by changing the following parameters:

— **C**: the number of cores (e.g., 4, 8, 16, 32), considering one core per tile

— **L2(xC)**: sharing of L2 cache, where C represents the number of cores sharing one L2 cache. For instance, the L2 is (x1) in the baseline architecture since each core has one L2 cache. An L2(x2) indicates that two cores are sharing the L2. The size of L2 for each core is set to 256KB; therefore, in L2(x2), two cores share an L2 with 512KB.

— **L3(xC)**: sharing of L3 cache, where C represents the number of cores sharing one L3 cache. It adopts the same rule used in L2. For instance, the baseline architecture

(assuming that there are four cores in total) adopts an L3(x4) configuration.

— **L2 size**: the size of the L2 cache dedicated for each core. When a core shares the L2 cache with another core, i.e., L2(x2 or more), the final size of the L2 cache will be multiplied by the number of shared cores.

— **L3 size**: same rule than L2 size.

The multi-core was simulated with the Sniper multi-core simulator [31]. The Nehalem cores are by default provided within Sniper distribution. The Sniper core model and cache hierarchy are validated against the actual Xeon processor using Splash2 benchmarks. Sniper takes as input configuration files that allow the user to set parameters such as cache sizes, cache sharing, number of cores, core frequency, among many others.

Next, in the experimental setup section, we present further details about the multi-core setup simulated on Sniper.

3.2 The Cache Impact on Dataflow Applications

This section presents the experiments evaluating the cache limits for dataflow applications. The first subsection describes the experimental setup. The remaining subsections address the analysis of application's performance varying the following parameters: C, L2(xC), L3(xC), L2 size, and L3 size.

3.2.1 Experimental Setup

Hardware Setup

The experimental setup adopts the multi-core model described in subsection 3.1, configured on Sniper. Table 3.1 presents the hardware setup. These parameters are based on the real Xeon X5500 multi-core.

To evaluate the number of cores and cache sharing we created 22 multi-core cache configurations, varying the parameters C, L2(xC), and L3(xC). Figure 3.2 shows graphically the reasoning behind these configurations. Each configuration is a black spot in the figure. The configurations can be divided into 4 groups (different background colors in the figure) according to the number of cores ($C = 4, 8, 16, 32$) in which a given configuration

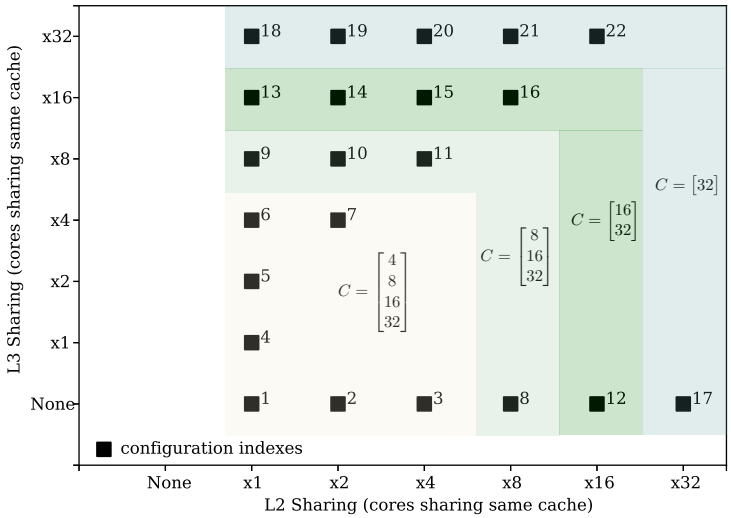


Figure 3.2 – Overview of the reasoning behind the 22 cache configurations adopted in the experiments. **C** = number of cores simulated for each configuration.

Table 3.1 – Experimental setup: Hardware model settings.

Core Model	Intel Xeon X5550 4/8/16/32 @ 2.66 GHz (base clock)				
L1-I Cache	32KB	8way	1 cyc. tag lat.	4 cyc. data lat.	LRU
L1-D Cache	32KB	8way	1 cyc. tag lat.	4 cyc. data lat.	LRU
L2 Cache	256KB	8way	3 cyc. tag lat.	8 cyc. data lat.	LRU
L3 Cache (LLC)	8MB	16way	10 cyc. tag lat.	30 cyc. data lat.	LRU

cyc = cycles; lat = latency; LRU = Least Recently Used.

was simulated. Note that the 22 configurations were not simulated for each C configuration. The minimal C evaluated for each configuration is dictated according to the sharing factor of the LLC. For instance, we do not evaluate a system with 4 cores for config. 9 (which have L3(x8) as LLC), since it is unfeasible because the L3 sharing (LLC sharing) requires at least 8 cores to meet the sharing factors of L3(x8).

The L2 sharing comprises configurations from L2(x1) up to L2(x32), with most of them (36%) addressing a private L2 cache (since this L2 design choice is found in real architectures like Xeon Nehalem and AMD K10). Some configurations are unrealistic, specially those that have a big L2, as the case of configurations 8, 16, 21, where L2 = 2MB; configurations 12, 22, where L2 = 4MB; and configuration 17, where L2 = 8MB. However, our goal is to address the trend in multi-core processor design, which features always bigger L2 caches.

The L3 sharing also adopts a very heterogeneous configuration set, including no L3

Table 3.2 – Experimental setup: Dataflow applications benchmark profile.

Application	Actors	PREESM #FIFOs	PREESM FIFOs size	Memory copying[a]	
				PREESM	Actors
Stabilization	30	607	0.92 MB	21 MB	0.2 MB
Stereo	36	811	29.09 MB	5 MB	13 MB
SIFT	77	2183	188.6 MB	12 MB	308.6 MB

(a) sum of all copied memory using the `memcpy` procedure.

(e.g. configuration 1), one private L3 cache (e.g. configuration 4), up to 32 cores sharing the same L3 (configuration 18-22).

The number of memory controllers is equal to the number of LLC. For instance, configuration 6, running with 8 cores, has two L3 shared by 4 cores (L3(x4)). Therefore, this configuration has two memory controllers (one for each L3).

Although the results achieved are based on Xeon architecture, the presence of 22 different hardware configurations, varying the core count and cache sharing and size, helps to project the behavior of the benchmarks in architectures different from Xeon, especially those that adopt similar cache organizations.

Application set

Table 3.2 (1st column) lists the applications benchmark addressed in this work. We adopt three real applications named Stabilization, Stereo, and scale-invariant feature transform (SIFT), taken from PREESM repository [112]. Stabilization is used for video stabilization. Its principle is to compensate for the movements of a video recorded with a shaky camera. The main two steps of this process consist of tracking the movement of the image using image processing techniques and creating a new video where the tracked motion is compensated. The input video adopted in experiments comes from PREESM's github repository [112] and has 40.9 MB of size with a resolution of 360x202 pixels.

Stereo is a computer stereo vision application that extracts 3D information from images. Stereo matching algorithms are used in many computer vision applications to process a pair of images, taken by two separated cameras at a small distance, and produce a disparity map that corresponds to the 3rd dimension (the depth) of the captured scene. Stereo matching algorithms and their implementations are still heavily studied as they raise important research problems [58]. The two input images [112] adopted in our experiments

have the size of 506.3 KB with a resolution of 405x375 pixels.

SIFT is used to object recognition in cluttered real-world 3D scenes [87]. The extracted features are invariant to image scaling, translation, and rotation, and partially invariant to illumination changes and affine or 3D projection. The application behavior shares a number of properties in common with the responses of neurons in the inferior temporal cortex in primate vision. The input image [112] used in SIFT has a size of 512 KB with a resolution of 800x640 pixels, with 4 levels of parallelism and 1400 keypoints.

These three applications are specified through the PREESM framework, which is responsible for the code generation, actors' scheduling and mapping, as discussed in Section 2.4 p. 64.

Table 3.2 highlights that the applications have heterogeneous memory requirements. Specifically, the 4^{th} column details the sum of PREESM FIFOs size, which can be understood as the memory footprint of inter-actor communication. SIFT is memory bounded and has high synchronization demands (high number of actors and FIFOs), Stereo is computational and memory bounded, and Stabilization is computational bounded but with low memory and synchronization demands. The heterogeneous memory requirements lead to different cache locality and memory footprints, making such applications appropriated candidates for the evaluation of cache impact intended in this work.

We use the optimal scheduling and mapping decisions provided by PREESM [110], which is focused on workload balancing. The memory allocation adopts advanced memory optimization proposed in [39], which considerably reduces the applications' memory overhead. The selected memory allocation uses the *FirstFit* algorithm with *MixedMerged* distributions and *none* data alignment. These features were selected because they have presented the lowest memory footprint at the same time that they are suitable to the target multi-core architecture used in this work. After the generation of C code by PREESM, the applications were compiled using GCC v7.5.0 optimization -O2 (default optimization adopted by PREESM), and simulated on Sniper.

3.2.2 Number of Cores – C

Figure 3.3 shows the application execution time (time for the application to complete the execution of one loop), for Stabilization (a), Stereo (b), and SIFT (c). The x-axis contains groups of bars, where each group represents one configuration (only the ones that support C varying from 4 to 32 were shown), and each bar represents a different C to that configuration.

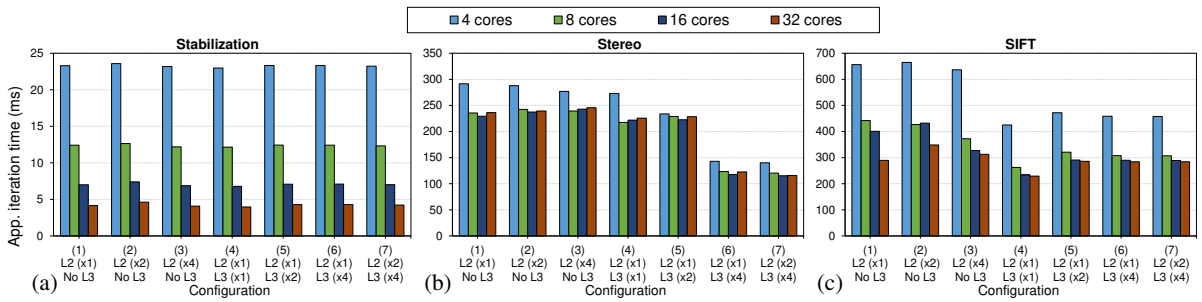


Figure 3.3 – Application iteration time over different number of cores for three applications: (a) Stabilization, (b) Stereo, (c) SIFT.

The main evaluation to be extracted from these results is related to scalability with the number of cores C. It is possible to observe that Stabilization presents some execution time improvement according to a higher C. It improves the execution time on average by 46% from 4 to 8 cores, 43% from 8 to 16 cores, and 39% from 16 to 32 cores. However, the same effect does not occurs to Stereo and SIFT. Indeed, we observe a moderate or even worst improvement in $C \geq 16$, with Stereo presenting an execution time of -22%, -1.3%, +2.6%, for an increase in C of 4 to 8, 8 to 16, and 16 to 32, respectively.

As represented in Table 3.2, Stabilization (which is scalable) has smaller FIFO sizes than Stereo and SIFT. The FIFOs of Stabilization fit in the cache whereas FIFOs of Stereo and SIFT do not and also suffer from lower data access locality.

It is also possible to observe that there are different performances among the configurations of the x-axis. Such performances are impacted by the different L2 and L3 sharing configurations. The next two subsections enter into details about the impact of L2 and L3 sharing.

3.2.3 L2 Sharing

Figure 3.4 presents a comprehensive evaluation of the L2 sharing impact over the execution time, L2 miss rate, and L2 miss rate for the three applications. The left y-axis of each plot represents the application iteration time, the right y-axis represents the miss rate, and the x-axis represents the configurations.

Each application has four plots, one for each simulated C. As the purpose is to evaluate the results only while varying L2 sharing, the plots have fixed L3 sharing parameters according to the maximum number of cores (as well as in the Xeon architecture).

The L2 miss rate decreases for all applications, more sharply for Stabilization (-59%),

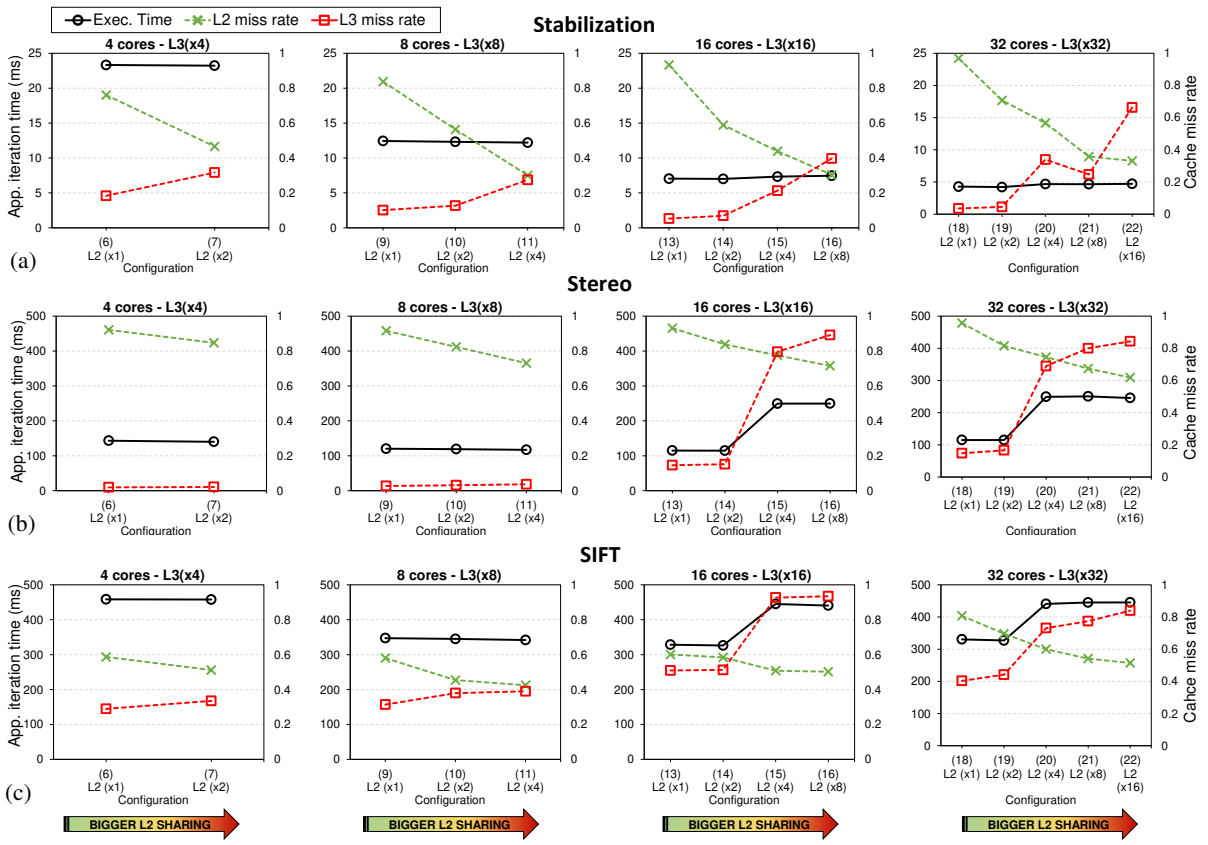


Figure 3.4 – L2 sharing evaluation for three applications. (a) Stabilization, (b) Stereo, (c) SIFT.

and less significantly for SIFT (-23%), and Stereo (-22%), considering the average between the leftmost configuration and the rightmost configuration. This decrease in L2 miss rate happens because a high L2 sharing increases the probability of an actor to share a FIFO inside the same L2 that is being shared with another actor (without the need to retrieve the data at the L3 cache level). The decrease is less significant in high memory demand applications – as SIFT and Stereo – since they naturally require more memory than Stabilization.

The L3 miss rate increases for all applications according to the higher L2 sharing. Such an increase makes the L3 reach high miss rates of 84.3% for SIFT, 84% for Stereo, and 66.32% for Stabilization in configuration 22. Again, the memory demands of each application play an important role to stress the cache. The number of L3 accesses helps to justify this L3 miss rate increase. With a more shared L2, the L3 accesses consequently decreases, reaching, on average of -39.7% for Stabilization, -32.3% for Stereo, and -17.6%

for SIFT.

This makes the L3 lose spatial locality and increasing its miss rate, which transfers the data access to DRAM level and delays the execution time.

The execution time remains constant for Stabilization regardless of higher L2 sharing. For Stereo and SIFT, it remains constant for $C = 4, 8$, but for $C \geq 16$, the execution time starts to increase from L2(x2), reaching up to +56% of increase for Stereo and to +17% for SIFT L2(x32). This increase in execution time is attributable to the significant increase of the L3 miss rate compared to a not-so-high decrease of the L2 miss rate, which generates miss penalties from both sides (L2 and L3 caches).

In summary, increasing L2 cache sharing is not beneficial to dataflow applications, specifically for those applications which demand more memory as in the case of Stereo and SIFT. This is in compliance with the cache design choices of some processor architectures as Intel Nehalem and AMD K10, which use private L2 caches. As can be observed from the results, assigning to each core a private L2 reduces the execution time since this allows a more balanced rate of L2 and L3 misses, which reduces cache contention earlier avoiding data to be fetched in a lower level of caches or even DRAM.

3.2.4 L3 Sharing

Figure 3.5 presents a similar set of plots of L2 sharing analysis, but now about L3 sharing. The L2 sharing is fixed in L2(x1) since the previous subsection has shown that this is the best L2 sharing configuration.

The results show three trends: (*i*) L2 miss rate remains constant; (*ii*) L3 miss rate decreases significantly according to the increasing of L3 sharing; and (*iii*) the execution time can benefit from a higher L3 sharing.

Regarding the L2 miss rate, it is expected that it remains constant since the L2 was not changed. Regarding the L3 miss rate, it decreases significantly for all applications according to higher L3 sharing, reaching a miss rate in the L3(xC) of, on average, 9.3% for Stabilization (reduced by 87.34%), 8.4% for Stereo (reduced by 87%), and 37.8% for SIFT (reduced by 38%). This result is expected since a higher L3 sharing allows all application data to fit in the L3 cache (note that SIFT presented the lowest improvement due to its higher memory demands). Consequently, the execution time also benefits from this L3 miss rate decrease, specifically for the applications with higher memory demands such as Stereo and SIFT.

In summary, increasing L3 cache sharing is beneficial to those dataflow applications

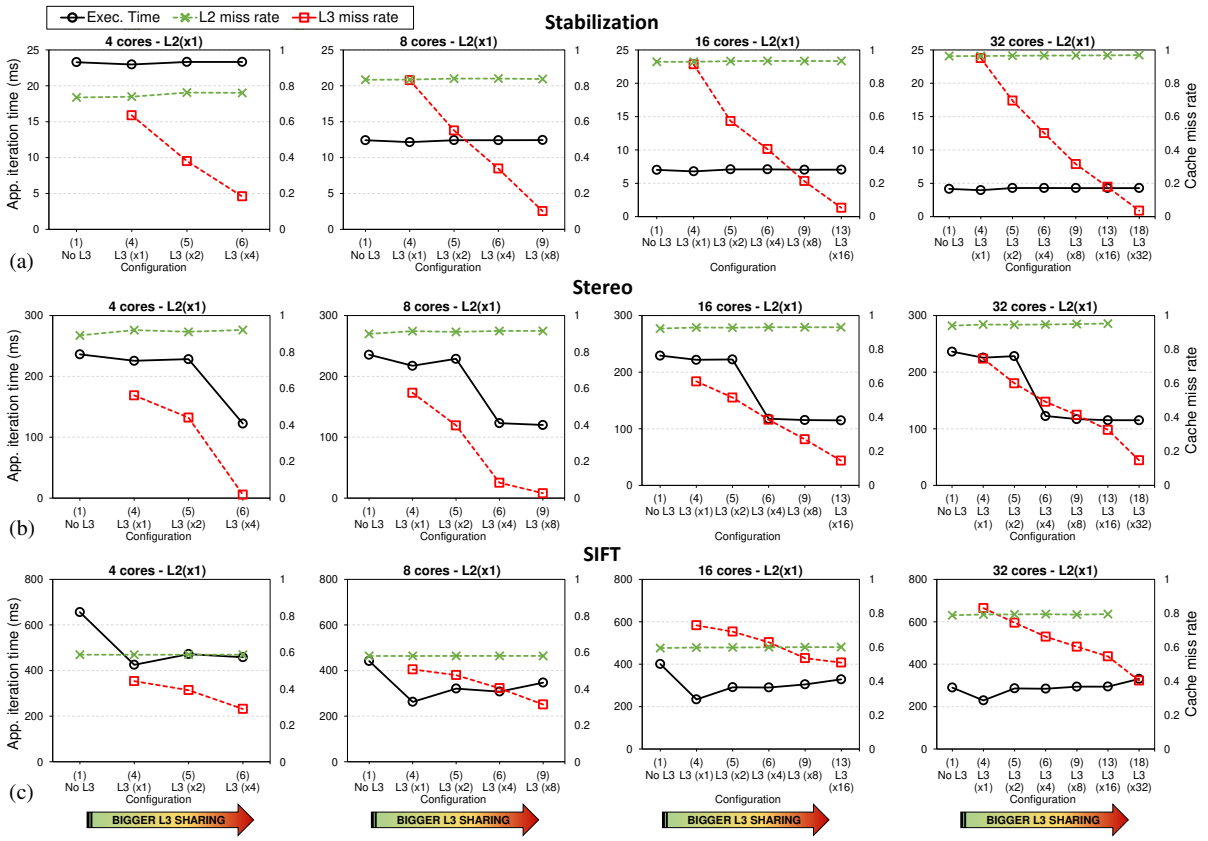


Figure 3.5 – L3 sharing evaluation for three applications. (a) Stabilization, (b) Stereo, (c) SIFT.

which fit in L3. A single L3 cache is slower but larger, allowing it to store all application data in it.

3.2.5 Cache Size

In the previous L2 and L3 sharing analysis, it was possible to conclude that an L2 private and an L3 shared by all cores presents the best results related to application speedup and L2/L3 miss rate. To the cache size evaluation, we keep this sharing configuration, and changed only the size of L2 or L3 per core, creating 15 new cache configurations (3 varying L2 size $\times$ 5 varying L3 sizes). Besides, the evaluation only addresses configurations with 32 cores, since lower core count have presented the same trend.

Figure 3.6 shows the results varying the L2 size (256KB, 512KB, and 1MB) at x-axis. The left y-axis represents the application iteration time, and the right y-axis represents the cache miss rate. Each plot represents one application, with each one having 3 sets of

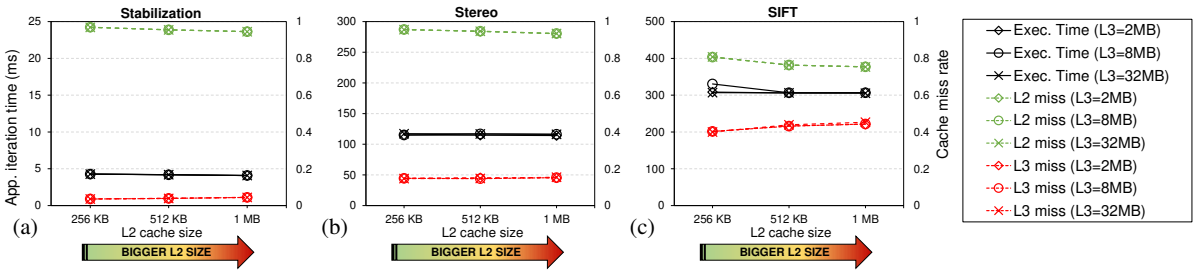


Figure 3.6 – L2 cache size comparison varying L2 size over multiples L3 sizes. (a) Stabilization, (b) Stereo, (c) SIFT.

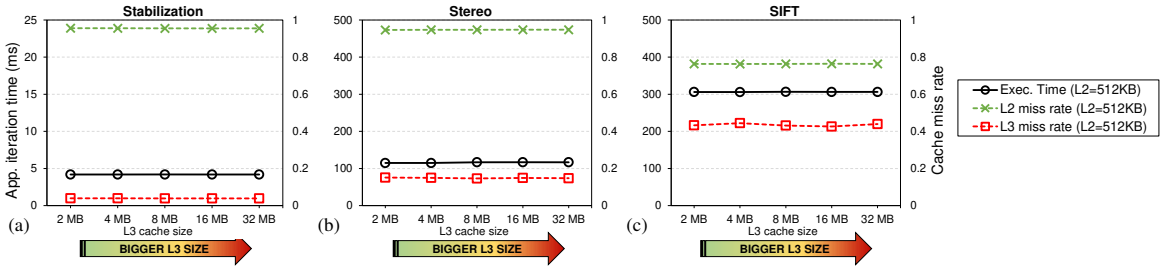


Figure 3.7 – L3 cache size comparison varying L3 size with an L2 private of 512KB. (a) Stabilization, (b) Stereo, (c) SIFT.

results representing different L3 sizes.

It is possible to observe that the increase in L2 and L3 size has a low influence on the L2 and L3 miss rate for all applications. The execution time has a small reduction according to higher L2 sizes, however, this value is insignificant, representing an average reduction from the lower L2 size (256KB) to the higher L2 size (1MB), of -0,49% for Stereo, -1.76% for SIFT, and -4.62% for Stabilization.

The results varying the L3 sizes follow the same trend observed for L2. Figure 3.7 shows an example with the L2 size fixed in 512KB (other L2 sizes present very similar behavior). It is possible to see that both L2 and L3 cache misses remains stable, and with an insignificant reduction in the execution time (not better than -0.26% for all applications).

In summary, increasing the L2 and L3 sizes does not guarantee an automatic improvement of the execution time for dataflow applications. In such a case, when a higher amount of memory resources cannot provide speedup to the application, other aspects must be taken into consideration. In particular, at the software level, mapping and scheduling algorithms can efficiently use memory resource availability and improve the parallel workloads

of the application.

3.2.6 Summary of Findings

Bigger caches are not always better with dataflow; increasing the number of cores, cache levels, size, does not guarantee a faster application execution. This finding is especially significant for working sets that demand more than the total cache size.

The next items summarize the finding for each analysis:

— **Number of cores**: increasing the number of cores does not guarantee automatic improvement of the execution time, since the overhead of cache protocols and required synchronization does not allow applications to increasingly speed-up, specifically the ones with more memory demands.

— **Cache sharing**: Despite our expectation based on advantage of sharing middle cache for FIFO sharing, reducing L2 sharing and increasing L3 sharing was the most beneficial configuration for the addressed dataflow applications.

— **Cache size**: increasing the L2 and L3 sizes have an insignificant effect on the adopted dataflow benchmarks.

One intriguing finding is that L2 private and L3 shared by all cores was the configuration that presented the best results related to application speedup and L2/L3 miss rate. While this conclusion can sound similar to what Intel came up with some years ago, justifying its current cache organization with L3 shared by all cores, it was not so straightforward from our point of view. First, our focus was to evaluate the impact specifically for dataflow applications, research that, to the best of our knowledge, was not addressed yet. Secondly, our initial hypothesis was that when two actors – sharing the same FIFO – are mapped on different processors that share an L2 cache (increased sharing factor), this will improve performance due to the reduction in the coherence traffic and the L2 miss rate reduction. This behavior is supported by the results (Figure 3.4). However, this leads to a higher miss rate for L3, which has higher penalties than L2, and consequently, has a higher influence on the execution time, as shown in the case of the three applications studied (Figure 3.5).

3.2.7 Final Remarks

This section presented a broad analysis of the impact of cache hierarchy configuration over static dataflow applications. In total, 37 different cache configurations (resulting in

213 simulations with 3 real applications) were considered to evaluate variations in core count, L2/L3 sharing, and L2/L3 sizes. From this analysis, it is possible to conclude that bigger is not always better in terms of core count, L2 sharing, and L2/L3 size, since other aspects like efficient parallel workload division and computation/communication profile can prevent the application to benefit from more cache memory resources. This analysis shows that private L2 and L3 shared among all cores provide the best results in terms of application speed-up and L2/L3 cache miss for the adopted dataflow applications.

Table 3.2 shows that PREESM uses extensively memory copying mechanisms for FIFO handling. Some memory copying is expected in a dataflow design; however, memory copying is done to the degree that negates the cache hierarchy benefits. Therefore, alternative approaches must be investigated to allow reducing memory copies penalties at runtime. The next section details the research made in this direction.

3.3 Dynamic Memory Management Techniques

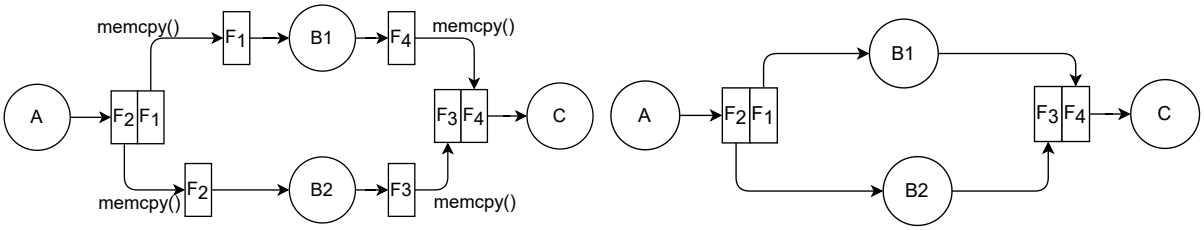
This section presents the evaluation of two dynamic memory management techniques and their impact when used in the context of static dataflow applications. We use static dataflow applications since they represent the more conservatively adapted class of dataflow programs to our techniques. However, the memory management techniques studied are also applicable for other classes of dataflow application.

These techniques are Copy-on-Write (CoW) and Non-Temporal Memory (NTM) copying. They are not novel in their principle, CoW is a well-known approach supported by Linux OS by the mmap() syscall [27], and NTM is essentially a direct RAM-to-RAM copy, supported in some Intel processors [65]. The novelty here is to exploit opportunities of using such techniques in dataflow frameworks, and quantify how much they can improve the applications execution time and system energy by saving memory transfers.

3.3.1 Motivation to Use CoW and NTM

Figure 3.8a shows the graph of a dataflow application that includes a fork and a join actor. Actor *A* produces data *d1* for actors *B1* and *B2*, which access and process them to generate a data for actor *C*, which receives them through a join actor and then merges both data and computes the application output.

Usually, a memory copy is used to copy the data received in the input port to the



(a) Classical dataflow fork-join application (b) Memory optimization proposed by [39]

Figure 3.8 – Communication overview among actors of a dataflow-based fork-join application.

output port in *Fork/Join*, broadcast, and *Roundbuffer* actors.

Each of these special kinds of actors naturally requires to allocate and use their own dedicated input FIFOs and output FIFOs to store the data. These input and output FIFOs can be allocated in separate memory spaces whose address spaces do not overlap with each other. These are called non-overlapping memory spaces. Desnos et al. [39] identified that there are opportunities to overlap these input and output buffers and therefore reduce and optimize the memory usage of these actors. Figure 3.8 illustrates the effects of this optimization and where in Figure 3.8(b) the memory copy for fork and join actors are removed and their input and output buffers are overlapped, which reduces the memory usage. However, the optimization proposed by Desnos et al. [39] is only possible if the actors *B1* and *B2* do not use the output of the fork operation as a temporary memory space to store the intermediate results of their computation. Since, the application is specified with dataflow MoC, the input to an actor is expected to be used and then discarded. As such, the allocated memory space for its inputs can be reused for the intermediate calculations to save memory as even described in page 30:5, Fig. 4 of Desnos et al. [39]. The reuse of input buffer for intermediate results can save significant amount of memory and reduce the memory footprint of the dataflow application.

In summary, the application developer can inform the framework which buffers can be merged in the same memory space, resulting in the graph presented by Figure 3.8b. Thus, some memcpy are avoided, helping to significantly save memory footprint and execution time.

However, such design-time approach only works assuming two conditions: (*i*) At design time, the framework must be fed with the designer's information related to actors' memory access patterns. However, it is not possible when application behavior is dynamic (changes at runtime). (*ii*) it only works for buffers that are known to be read-only over all the actor

lifetime, as the case of buffer *F1* of actor *B1* of Figure 3.8a. If the actor – due to a branch in its algorithm flow – chooses to write in *F1* buffer space, these memory reuses cannot be adopted. Even if the actor has a probability of less than 1% to write in this buffer, the memory reuse cannot be applied since it is fundamentally a design-time exploration technique.

Thinking about how to fulfill this lack, our idea is to investigate two dynamic memory management techniques which are CoW and NTM. Memory management for static data flow applications that are focused on in this chapter can be done statically at compile time or dynamically at runtime. Our approach is designed to be used at runtime, and have the potential to avoid unnecessary memory copies (CoW), and cache thrashing (NTM).

Cache thrashing happens when a current task/actor is running and then swapped for a different task/actor by the scheduler. The new task starts using the cache and replaces the cached data of the previous task/actor. When the previous task is swapped back in, the cache does not contain its data and loses the benefit of caching. As such, memory intensive tasks that quickly fill the data cache might thrash each others' cached data.

In the next subsections, we present the details of each one and how they were implemented in our multi-core model, as well as, the evaluation of their drawbacks and benefits.

3.3.2 Copy-on-Write (CoW)

Our proposed CoW technique complements and enhances the work by Desnos et al. [39]. We apply their compile optimization first that reduces the memory copies. However, they need to perform compile-time range analysis and ensure that consumer task will not write to the shared input buffer as a temporary space for intermediate results. This compile-time analysis hinges on pointer alias analysis that is inherently conservative to ensure correct execution of the program. CoW is a dynamic memory management technique that does not initially copy the input buffer data. If the consumer task does not write on the input buffer, no data is copied. However, if the consumer initiates a write on the buffer, a memory copy is initiated and a new memory is allocated. It is true to that dynamic memory allocation can incur some latency. However, the benefits comes from the fact that our technique does not depend on conservative alias analysis at the compile time and can cover more cases while minimizing the memory copies using runtime information.

Figure 3.9 details the CoW concept. The principle of CoW is simple. It consists in

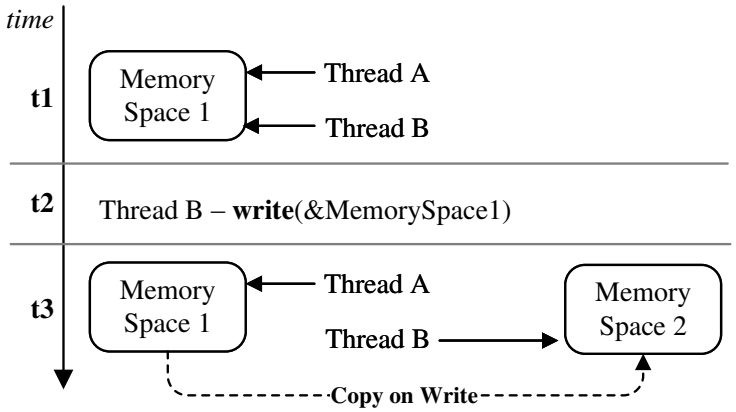


Figure 3.9 – Principle of the Copy-on-Write (CoW) mechanism.

Table 3.3 – Core change for supporting the CoW mechanism, assuming: (1) `src_buffer` is the source buffer, (2) `dst_buffer` is the destination buffer, (3) `copy_length` is the copy length, (4) `shm_open_fd` is a file descriptor created with the `shm_open` system call.

Original Code	CoW mechanism
`memcpy(dst_buffer, src_buffer, copy_length);`	`void *dst_buffer = mmap(NULL, copy_length, PROT_READ | PROT_WRITE, MAP_PRIVATE, shm_open_fd, 0);`

allowing two or more threads (actors in our case) to share to the same memory space. When one thread attempts to write in that space a new memory space with same size is dynamically created, bringing the data with it (a copy on write). It thus prevents the writing thread from overwriting the data in the first memory space [27]. Figure 3.9 depicts at time $t1$ thread A and thread B pointing to the same memory space 1. At the time $t2$, thread B writes in the memory space 1. At the time $t3$, the OS detects this write and makes a copy of the memory space, creating the memory space 2 and making thread B point to it. Now, any data written/read by thread B will be placed/accessed in memory space 2.

This functionality can be implemented in a dataflow application by making the buffers involved in a given operation (like a fork, join, and broadcast) point to the same memory space after the producer actor writes data in this space. If the destination buffer receives a write attempt by any actor, a CoW happens, preserving the original values of the buffer to the consumer actor.

The core code change to support CoW is shown on the right side of Table 3.3 as a single line. Initialization and termination has been omitted for this example. The CoW mechanism is achieved by mapping all destination buffers (named `dst_buffer`) into the same physical address, which is referenced by file descriptor `shm_open_fd`. This latter

Table 3.4 – Core change for supporting the NTM mechanism, assuming: (1) `src_buffer` is the source buffer, `dst_buffer` is the destination buffer, `copy_length` is the copy length.

Original Code	NTM mechanism

`memcpy(dst_buffer, src_buffer, copy_length);`	`for (i = 0; i < copy_length/4; i++){`
	`_mm_stream_si32(dst_buffer[i], *(src_buffer[i]));`
	`}`

address was initialized by the `shm_open` procedure. Besides, we map the region as private (`MAP_PRIVATE`) with read and write permissions (`PROT_READ | PROT_WRITE`). The combination of these two flags will create a new copy of the physical address when a write occurred in the memory area (in other words, a copy-on-write). Finally, we allow the OS to decide the virtual address of this new buffer by passing nil (`NULL`) as the first parameter to `mmap`.

The CoW procedure is typically handled by the OS kernel. Unfortunately, the Sniper simulator has limited operating system modeling capabilities to evaluate kernel-based strategies [31]. Thus, for our experiments, we used a combination of user- and kernel-space interaction so that Sniper can account OS overhead accurately. Specifically, in our implementation a buffer is mapped to CoW without the `PROT_WRITE` flag. Not setting this flag makes sure that the buffer is not writable (does not have write permission). Any future attempts to write in the buffer will trigger an exception [1], which interrupts the causing thread. Then, a corresponding exception handler implemented in the user space changes the concerned memory page to use CoW. In regard to the code presented in Table 3.3, as described we change the capability of the memory regions to read-only (removing the flag `PROT_WRITE`), and install an exception handler to re-enable the write capability for this mapping.

3.3.3 Non-Temporal Memory (NTM) Copying

Non-Temporal Memory (NTM) copying is ideal for memory spaces known to be write-mostly and rarely used (i.e., poor temporal locality). This approach uses either (*i*) instructions that bypass the cache hierarchy or (*ii*) user space RAM-to-RAM DMA. In our proposal, the `memcpy` procedure is replaced by another procedure that uses either the aforementioned two techniques specialized for memory transfer in dataflow applications. Therefore, our proposal also relieves the CPU from executing the data transfers instruc-

1. For the exception, we use the `SIGSEGV` event, which is a synchronously-generated OS interrupt that is guaranteed to be delivered to the causing thread [64].

tions. Another benefit of employing this NTM mechanism is that cached data from other applications are not thrashed due to the copying required by any given application. NTM already exists in current processors, our work utilizes it for dataflow applications. For instance the x86 architecture offers NTM through (*i*) SSE extensions [65], and (*ii*) the I/OAT DMA engine that is also available in some Intel processors [78].

Since these approaches bypass the cache hierarchy, their operation is slower compared to `memcpy`. Intel shows that RAM-to-RAM achieves approximately half the speed of `memcpy` for large transfers ($\geq$ 8 MiB) but is many times slower for smaller transfers on x86 [78].

Table 3.4 shows the code change to use NTM instructions for Intel processors on Linux operating system.. The core code change to support NTM is shown on the right side of Table 3.4 as a for-loop structure. Initialization and termination has been omitted from this table. The procedure `_mm_stream_si32` is provided by Intel to call the appropriate assembly instruction for NTM operations. It copies 32 bits from a value (`src_buffer[i]`) to a given pointer (`dst_buffer[i]`). After the end of the for-loop, the data is copied to the area pointed by `dst_buffer`. Thus, the result is the same as calling `memcpy` but the related data will not be present in the caches if they were not already there before `_mm_stream_si32` is first called.

3.3.4 Results

This section presents the results about CoW and NTM using the three dataflow benchmarks, SIFT, Stereo and Stabilization, and the 22 configurations on Intel Xeon architecture.

Experimental Setup

All applications used in these experiments were generated using the PREESM framework (version 3.4), compiled using GCC v7.5.0 optimization -O2, and executed on Sniper simulator. The energy estimation is performed with McPAT [82], which is integrated into Sniper and provides reliable power and energy figures broadly used in state-of-the-art works [74, 114].

We have developed an algorithm implemented in Python script language, which has as input the generated code of PREESM and has as output the new application code using the CoW or NTM technique. This algorithm detects in the code the memcpy patterns,

Algorithm 2: Patterning detection of CoW and NTM memcpy

Input : src_o (application source code generated by PREESM),
t (technique: CoW or NTM),
ψ (minimum data size of a memcpy)
Output: src_t (application source code adopting CoW or NTM)

```
1  join_broad_set ← ∅;
2  broad_set ← ∅;
3  memcpy_list ← extract_memcpy(src_o);
4  if memcpy_list ≠ ∅ then
       /* Identify join-broadcast patterns                                  */
5      foreach mj ∈ memcpy_list do
6          if mj.type = JOIN & mj.destination ∉ join_broad_set & mj.size ≥ ψ
            then
7              foreach mb ∈ memcpy_list do
8                  if mb.type = BROADCAST & mb.source = mj.destination &
                    mb.size ≥ ψ then
9                      join_broad_set.insert(mj.destination);
10                     break foreach;
11                 end
12             end
13         end
14     end
       /* Identify broadcast-only patterns                                  */
15     foreach mb ∈ memcpy_list do
16         if mb.type = BROADCAST & mb.destination ∉ join_broad_set &
            mb.size ≥ ψ then
17             foreach mb ∈ memcpy_list do
18                 broad_set.insert(mb.destination);
19             end
20         end
21     end
22 end
23 if t = CoW then
24     src_t ← applies_CoW(src_o, join_broad_set, broad_set, memcpy_list);
25 else
26     src_t ← applies_NTM(src_o, join_broad_set, broad_set, memcpy_list);
27 end
```

which are candidates to be replaced by CoW or NTM. However, these two techniques can be combined and can be performed simultaneously. The algorithm is fully automatized, in the sense that it detects the memcpy patterns by looking for join-broadcast, and broadcast dataflow patterns [39]. The algorithm can also be tuned with a parameter ψ, which allows defining a minimum threshold in the data size of a memcpy operation. By using ψ, it is possible to eliminate small-size memcpy and only target the ones which transfer a large amount of data.

Algorithm 2 presents the method to detect the memcpy patterns which are candidates to be used in CoW and NTM. As input, the algorithm has three parameters: *src_o*: the application source code generated by PREESM; t: a flag that selects between CoW or NTM; and ψ: the parameter that indicates the minimum memcpy data size. Line 1 and 2 initialize two sets, called *join_broad_set* and *broad_set*, which will store the destination buffers' names of memcpys related to join-broadcast and broadcast-only, respectively. In line 3, the function *extract_memcpy* extracts from *src_o* all memcpy instances generated by PREESM, achieving the following information from each memcpy: data transfer *size*, *source* buffer, *destination* buffer, and the type (JOIN, BROADCAST, FORK, and ROUNDBUFFER [109]). The type is easily extracted due to a PREESM's characteristic in which it classifies the memcpy during its code generation, inserting its type as a comment in the line above each memcpy. All the memcpy instances are inserted in the list called *memcpy_list*.

Lines 5–14 identify join-broadcast patterns evaluating each element mj of *memcpy_list*. The condition of line 6 checks if the mj is a JOIN, if its destination buffer is not already in *join_broad_set*, and if the memcpy data size meets ψ.

Once this check is true, the algorithm advances to the phase (lines 7 and 8) to confirm that the destination buffer of JOIN operation is also involved in BROADCAST. Once the buffer matches a join-broadcast pattern, it is inserted in the *join_broad_set* in line 9.

Lines 15-22 identify broadcast-only patterns evaluating each element mb of *memcpy_list*. Line 16 tests if mb's type is BROADCAST. Another important verification is to check if the buffer is not in the *join_broad_set*, eliminating it if true. The same test of line 16 also seeks to eliminate the memcpys with a size lower than ψ. If all conditions are meet, the destination buffer is added to the *broad_set* at line 18.

The last part of the algorithm (lines 23–27) focuses in verifying the value of t, calling the respective function which will applies CoW (line 24) or NTM (line 26). These functions evaluate all memcpys from *memcpy_list*, selecting those one containing the buffers name

Table 3.5 – Memcpy profile addressed in CoW and NTM.

Application	ψ	**# memcpy**	**Total memcpy size**
Stabilization	200KB	1	0.21 MB
Stereo	400KB	5	2.4 MB
SIFT	400KB	8	24.4 MB

in *join_broad_set* and *broad_set*. The output of the algorithm is *src_t*, comprising the *src_o* with the matched memcpys replaced by the code presented in Table 3.3 and Table 3.4.

Table 3.5 details the values of ψ (2^{nd} column) used for each application. The table also details the number of `memcpy` addressed in CoW and NTM (3^{rd} column), and its respective total size (4^{th} column). The `memcpy`s are replaced by the procedures given in Table 3.3 and Table 3.4 for CoW and NTM, respectively.

Table 3.5 shows that SIFT has more available memcpy to be optimized. The threshold ψ was defined as 400KB for SIFT and Stereo. Stabilization does not have such a large memcpy, therefore we reduce the value of ψ to allow the algorithm to consider the higher memcpy of the application. To find the values of ψ, the applications are profiled with different ψ values at the design time and the one that corresponds to the best execution time is selected.

The next subsections present the results achieved by replacing the memcpy with NTM or CoW separately. We study these techniques separately in order to understand their individual contributions to the performance and energy gains.

Copy-on-Write (CoW)

Figure 3.10(a) presents the iteration execution time for all benchmarks using CoW. An average execution time reduction can be observed for Stabilization (-2%) and, most importantly, to SIFT (-10%), which reaches up to -15.8% for configuration 10. It is expected that SIFT benefits more from CoW since it has a large number of buffers used in memcpy compared to the other applications. On the other side, Stereo presents an average execution time increase of 1.3%. Stereo is known to be computation-intensive, and, therefore, the access to buffer mapped as CoW is less frequent than in Stabilization and SIFT, which makes the overheads of CoW (create shared memory and call of mmap()) overcome its benefits.

An energy reduction was achieved for all applications (-7.6% on average), with an

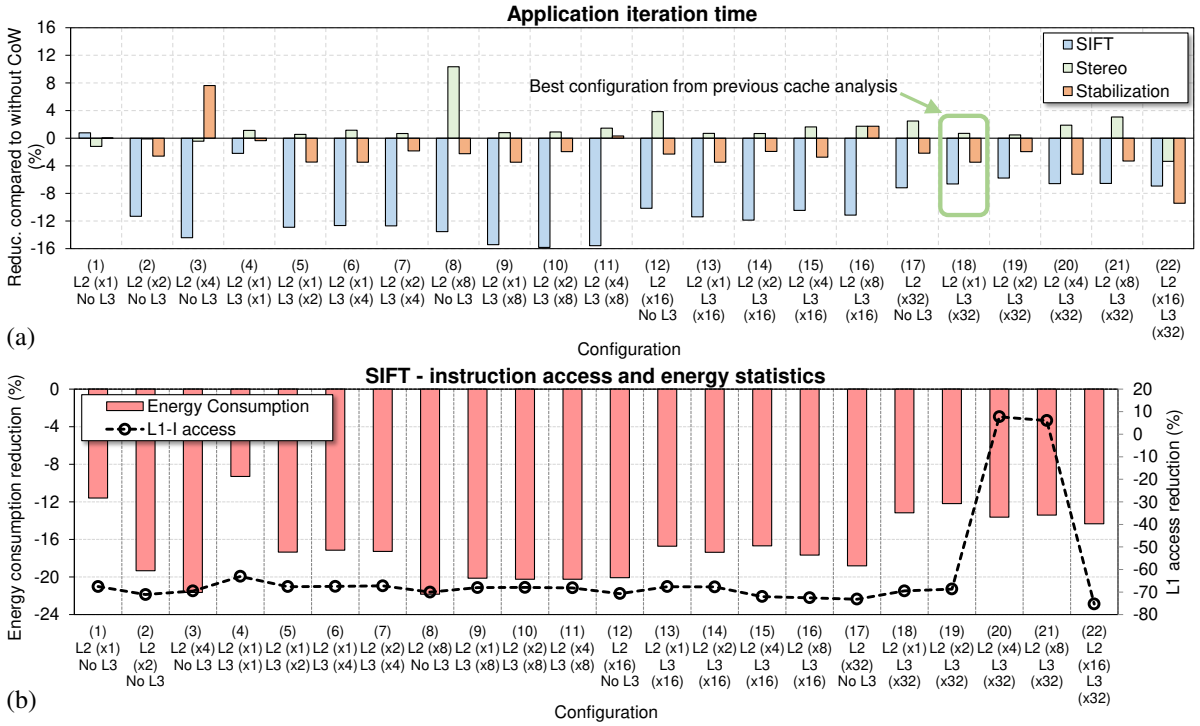


Figure 3.10 – Results using CoW. (a) Execution time evaluation. (b) Energy evaluation.

average reduction of -2% for Stabilization, and -1.3% for Stereo. Again, SIFT is the application that benefits the most from CoW regarding energy. Figure 3.10b shows an overview of energy consumption for SIFT (bar graph) to the 22 configurations. On average, the energy reduction was -16.8%, reaching the best result of -21.8% to configuration 8. Again, SIFT benefits from the CoW which allows data to be used without having to wait for the memcpy to complete. This behavior significantly affects the use of the CPU, which saves instructions in memcpy. This result can be observed following the dotted line of Figure 3.10b, which shows a significant reduction of L1-I (introduction cache) accesses of, on average, -62.3% (σ=3.4).

Figure 3.11 presents results for all applications in configuration 18. As expected, all applications have a reduction in the number of instruction access from the L1-I cache, which is justified by the saved memcpy instructions by using CoW. The instruction access gains progresses accordingly with the size of application's memcpy (as depicted in Table 3.5 (4^{th} column)), with Stabilization presenting -0.41% less L1-I access, Stereo -29.8%, and SIFT -46.5%. This effect impacts the energy consumption and execution time, especially for SIFT, which benefits more from CoW due to its larger memory transfer profile.

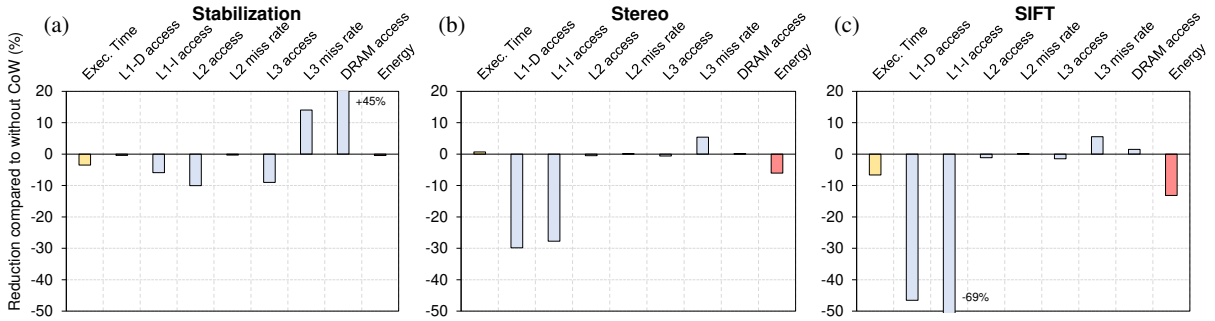


Figure 3.11 – Results for configuration 18 using CoW. (a) Stabilization. (b) Stereo. (c) SIFT

Non-Temporal Memory Copying (NTM)

Figure 3.12a presents the execution time for all benchmarks using NTM. It is noticeable that execution time is slightly reduced in most of the cases, reaching up to -1.9 % for Stabilization in configuration 21. The average execution time reduction was -1.9% for Stabilization, -1% for SIFT, and -0.3% for Stereo.

NTM also provided a small energy reduction, in average -0.84% (σ=0.7) for Stabilization, -0.2% (σ=0.5) for Stereo, and -1.03 (σ=1.1) for SIFT, reaching up to -2.7% for SIFT at configuration 16.

Figure 3.12b focuses on SIFT (high memory footprint application) and presents a perspective between the bars: L3 miss rate, execution time, and energy, with the lines that show the absolute number of DRAM accesses without NTM and with NTM. It is possible to observe that energy is reduced in most configurations, reaching up to -2,7% to configuration 16. The L3 cache miss was barely affected, presenting an average decrease of -0.13% with a slight DRAM increase compared to its respective version without NTM (+0.14%).

Figure 3.13 presents results for all applications on configuration number 18 (private L2 and L3 shared by all cores), which was the cache configuration that, in general, presented the best results considering speed-up and L2/L3 miss rate from previous cache analysis (see section 3.2). NTM has presented improvements for all applications on this configuration, specifically for SIFT and Stabilization. Note that, despite Stabilization has a low memory footprint, the execution time reduction is higher than SIFT and Stereo, at cost of more L3 miss rate. On another side, SIFT presents a modest execution time reduction, but also achieving reduction in all cache hierarchy, and specifically, in L1-D and L1-I access.

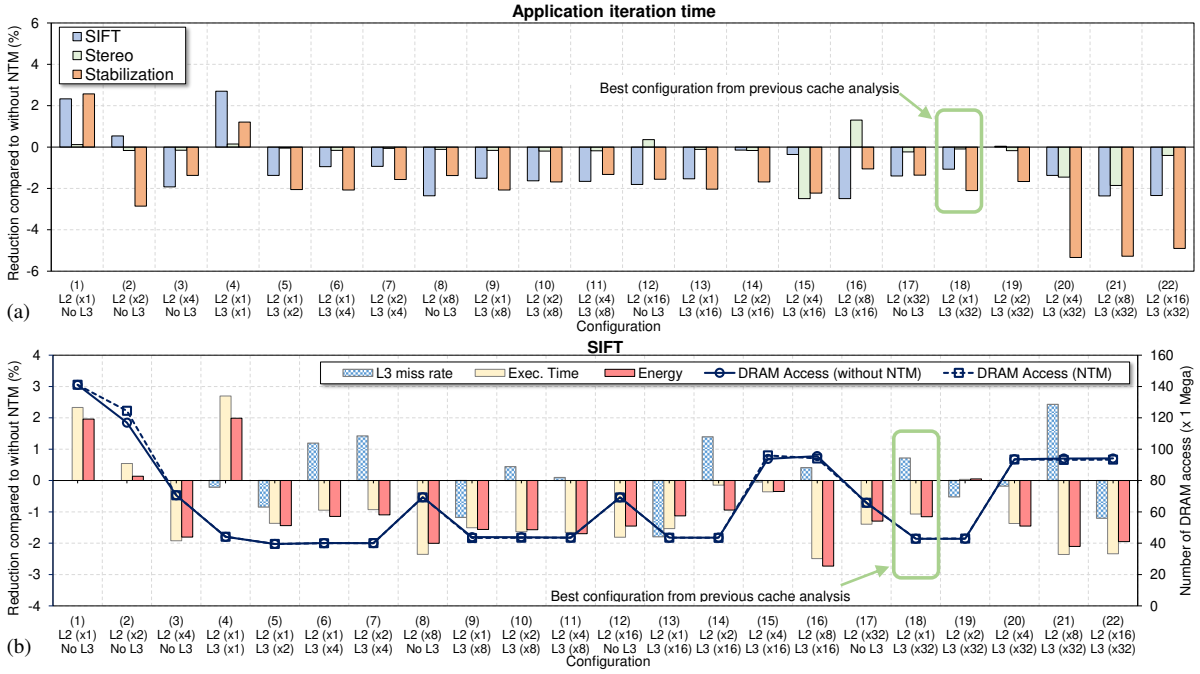


Figure 3.12 – Results using NTM. (a) Evaluation of execution time. (b) Detailed evaluation for SIFT application.

In summary of all results, it was possible to observe that NTM can improve the execution time and reduce energy, however, the gains were modest, not better than -1.9% in execution time and -2.7% in energy consumption considering all results.

3.3.5 Final Remarks

This section investigates the benefits of using copy-on-write (CoW) and non-temporal memory transfer copies (NTM) in dataflow applications. Results have shown that both techniques can contribute to improve execution time and save energy. NTM presents a modest reduction in execution time (up to -5.3%) and energy (up to -2.7%). CoW – specifically when used in applications with bigger memcpy transfers ($\geq$ 400KB) – shows noticeable reductions, achieving up to -15.8% in execution time and -21.8% in energy consumption. These techniques are complementary to static state-of-the-art memory optimization approaches like [39], acting at runtime to reduce cache thrashing (NTM) and unnecessary data movements (CoW) among dataflow actors.

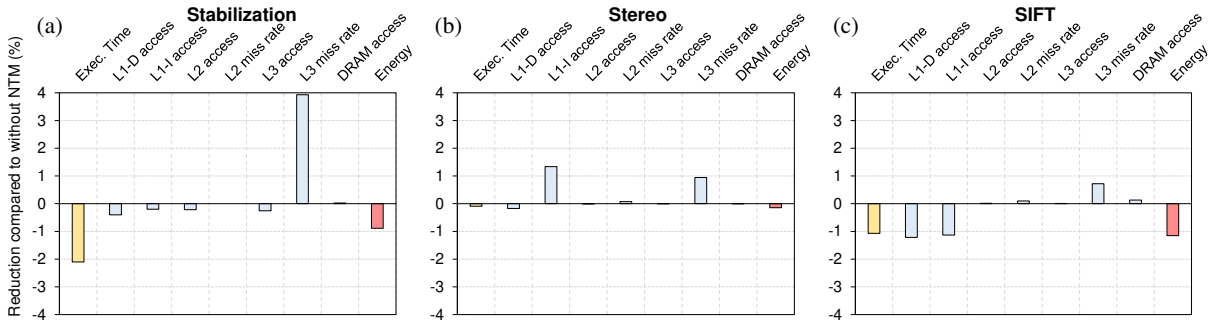


Figure 3.13 – Results for configuration 18 using NTM. (a) Stabilization. (b) Stereo. (c) SIFT

3.4 Conclusion

This chapter presented the first contribution of this thesis: the study of dataflow applications running on a SMP according to different cache configurations, and the automatic use of dynamic memory management techniques to improve the execution time by a better usage the cache hierarchy. The study of the impact of cache in SMP running dataflow application was missing in the literature and has been for the first time deeply explored. The main outcome of this study is that no real benefit for dataflow application can be expected with the current trend which is the increase of the number of cores and their cache sizes. The use of existing dynamic memory management techniques can help in some specific cases but there is no general case or systematic method to apply. It is not sufficient to really get rid of the intrinsic mismatch between cache and dataflow applications. A new approach is thus required to obtain significant improvements and this is what we present in the next chapter.

Chapter 4

NM4SMP: Notifying Memories for Symmetric Shared-Memory Multiprocessors

This chapter presents the second contribution of this thesis: a HW/SW co-design of an optimized near-memory computing device for synchronization of dataflow applications on SMPs called Notifying Memory for SMP (**NM4SMP**). It consists of near-memory hardware logic and its respective support at the software level through a library API and middleware. The study detailed in this chapter has been submitted to IEEE Trans. on Parallel and Distributed Systems and is currently under review. We propose a framework that comprises a complete tool set including the following items:

— Code generation framework integrated with PREESM [110],
— NM4SMP software library for dataflow applications,
— NM4SMP middleware for system executions,
— Simulation primitives in Sniper [31].

The contributions of this chapter can be subdivided as follows:

1. A HW/SW co-design, including library, middleware, and a hardware on-chip device to synchronize dataflow actors, called NM4SMP (Notifying Memory for SMP);
2. A Framework to integrate the NM4SMP co-design in a rapid prototyping methodology [110, 62], addressing dataflow modeling, code generation, and compilation of both static and reconfigurable dataflow applications.

4.1 Overview of the Proposal

Figure 4.1(a) presents an SMP platform with NM4SMP and Figure 4.1(b) shows a functional view of the NM4SMP mechanism. It is implemented close to the L1 cache

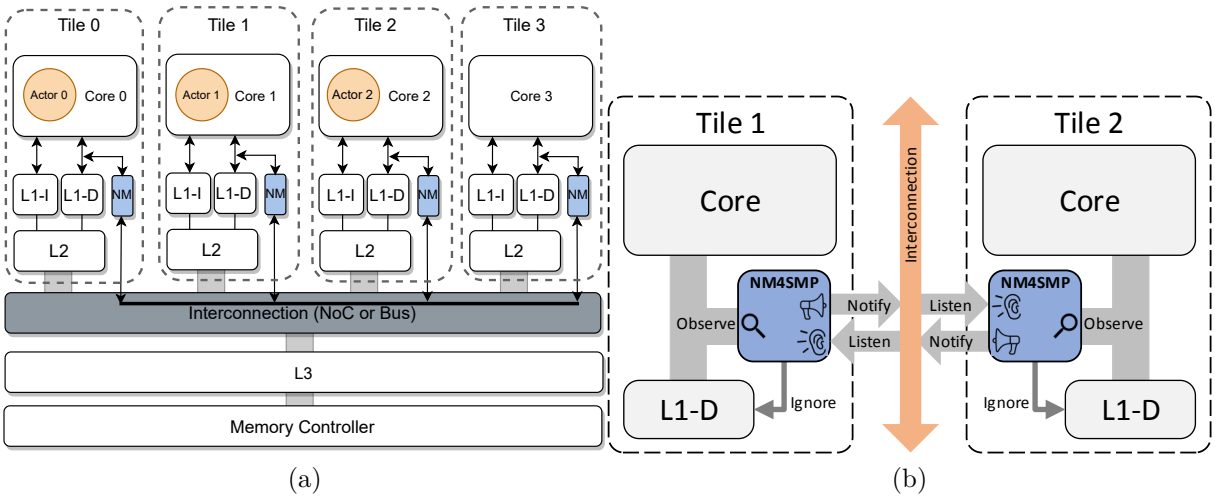


Figure 4.1 – (a) SPM with NM4SMP hardware module. (b) Overview of the proposed NM4SMP mechanism.

of each core. The NM4SMP hardware implements four processes: observe, notify, listen and report. The NM4SMP hardware is configured with the API and middleware and implements the firing rules of the actors. The NM4SMP *observes* the data and address buses between the CPU and L1 data cache(L1-D). When a producer actor generates the data, on the basis of the address of L1-D cache access, the module can detect and *notifies* the respective NM4SMP module of the consumer core by sending a *Notification* message. To prioritize the *notification* messages, they are communicated over the same existing interconnection network via a dedicated virtual channel. NM4SMP does not modify the core and does not modify the caching logic. It merely sits on the interface between the core and cache and also does not alter the critical path of load and store instructions.

On the consumer core (where the consumer actor is mapped) the NM4SMP module *listens* to the notification from the NM4SMP of the core who hosts the producer and updates the firing rule in its logic based on the *notification* message as it is received. *Only when all conditions of the firing rule of the consumer actor are satisfied, the NM4SMP reports the core* to start the execution of the consumer actor. Our **assumption** is that NM4SMP can boost the performance of dataflow application since it acts to remove synchronization overheads at both hardware and software levels. At the hardware level, NM4SMP eliminates the role of the cache to manage the synchronization primitives (lock values) and at the software level it avoids constant interruptions and context saving/restoring required to produce and to check each firing rule. Another expected positive side effect is to reduce cache thrashing by freeing the cache of dealing with the data used for synchronization.

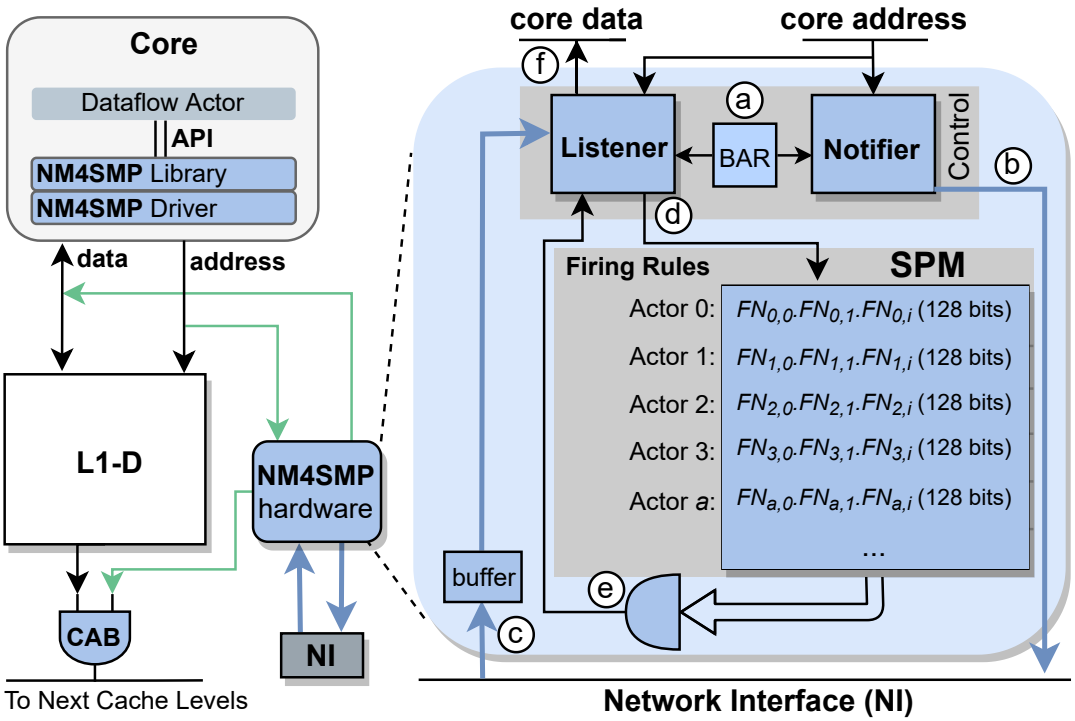


Figure 4.2 – Overview of NM4SMP software/hardware: new components are highlighted in blue. **BAR**=Base Address Register. **SPM**=Scratchpad Memory.

4.2 Notifying Memory for SMP (NM4SMP)

This section presents the NM4SMP design. We adopt a top-down explanation, addressing the NM4SMP impact at the user level and in sequence detailing its middleware and hardware implementation. Figure 4.2 shows an overview of the software/hardware stack assumed in NM4SMP. The actors communicate using API functions provided by the NM4SMP library. The NM4SMP library interfaces an NM4SMP middleware implemented as a user-level library or as a traditional driver (OS). The middleware implements the interface with the hardware NM4SMP module, configuring it at runtime.

4.2.1 NM4SMP Library

The role of the NM4SMP library is to abstract the notifying memory features to the actors through API functions. The library provides the following APIs:

— `send(prodID, consID)`: called when an actor `prodID` produces data in the FIFO to actor `consID`. `prodID` and `consID` are unique ID numbers ($ID \in \mathbb{N}$), assigned

to each actor of SMP.

— `receive(prodID, consID)`: called by `consID` to consume a data from `prodID`.

— `initialize(`a_{list}`,` fr_{list}`)`: called by a static application once during its initialization or by a reconfigurable application at the beginning of each new iteration. It initializes the NM4SMP by the list of actors (a_{list}) and their firing rule (fr_{list}).

Indeed, NM4SMP library implements a 2D-matrix used for the **F**IFOs **N**otification status, called FN. At the application level (library), each $FN_{a,i}$ flag represents whether data on input FIFO i is ready to be consumed by actor a.

An important remark is that the FN matrix is mapped to the Scratch Pad Memory (SPM) of the NM4SMP hardware module with the help of NM4SMP library (see the SPM of Figure 4.2).

A SPM is a high-speed memory used for temporary storage to hold a limited amount of data. It is employed to simplify the local access to data when the cache logic is unnecessary.

At application level, the *send* and *receive* functions are meant to write and read, respectively the elements of this matrix ($FN_{a,i}$s). Since the FN matrix has its addresses mapped to the SPM, the NM4SMP hardware module can observe the accesses to the FN and takes different actions according to them. It can send notification to the destination core to update the SPM or to check if a firing rule is satisfied then report the core. Indeed, all these actions are handled in different phases of NM4SMP state machine in source and destination cores.

At the hardware level, $FN_{a,i}$ is mapped on a 1-bit flag that represents whether the data is available in the input FIFO i of the consumer actor a or not. When all the flags for a given actor consumer a are equal to 1 (i.e. $1 = \{FN_{0,i}.FN_{1,i}.FN_{a,i}\}$), the consumer actor has its firing rule satisfied and it can start its execution in that iteration (see the Figure 4.2).

The *initialize* function initializes the FN matrix with the FIFO flags of the actors associated to them. For static dataflow applications, since the firing rule does not change at runtime, this function is called only once during the initialization part of the application. For reconfigurable dataflow applications, with dynamic firing rules, this function is called at the beginning of each iteration of the graph during the execution of the application.

4.2.2 NM4SMP Middleware

The NM4SMP middleware is a software abstraction layer from the hardware used to decouple the library from different architectures. Its goal is twofold:

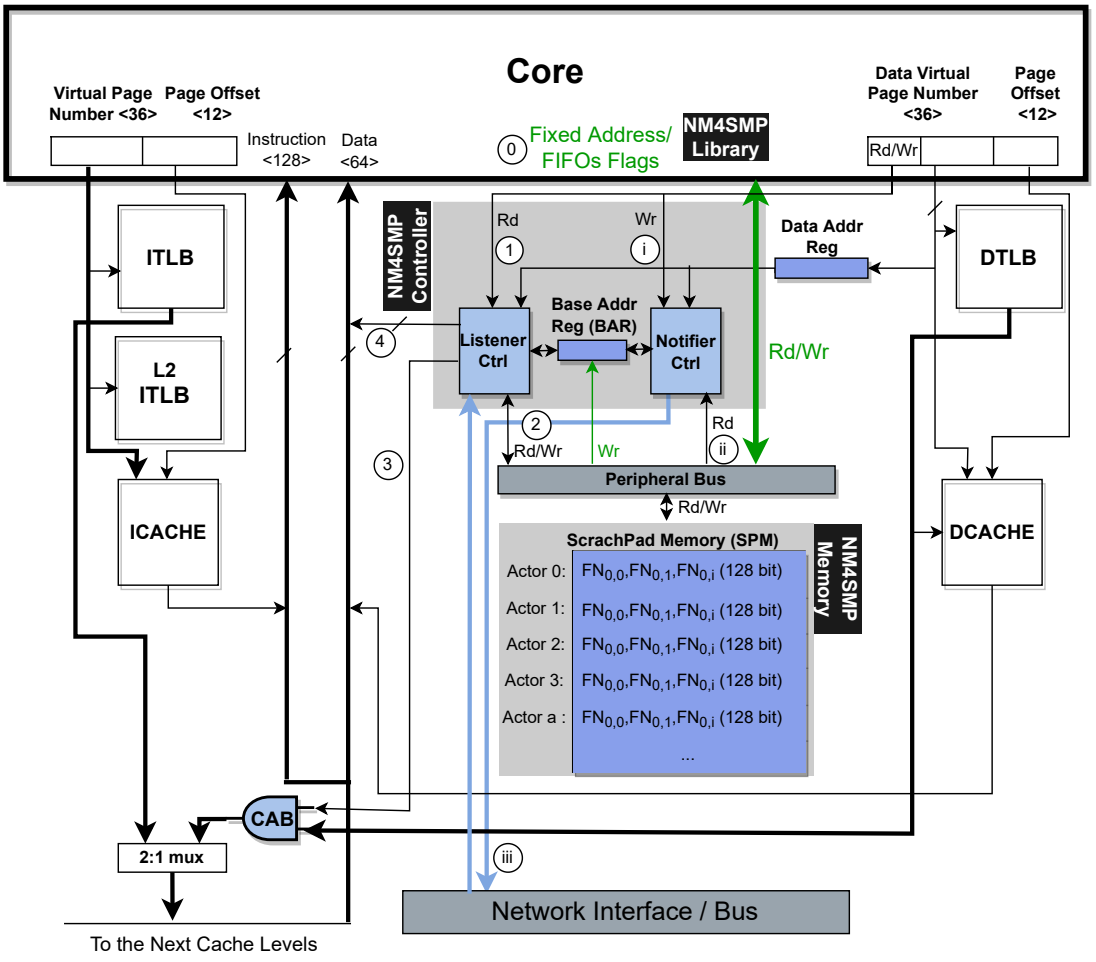


Figure 4.3 – A more detailed view of the NM4SMP hardware. The coloring highlights the components of the design.

(*i*) to configure the SPM of the NM4SMP hardware when the *initialize* function initializes the SPM at booting time with all the bits equal to 1 and then at compile/runtime of the application with FN matrix values that is 0 for each valid FN_i;

(*ii*) to configure the Base Address Register (BAR) of the NM4SMP hardware at booting time with the base address of FN matrix, so the hardware can observe exactly when the core is accessing it due to a call of *send* and *receive* functions (see subsection 4.2.3 for more details).

We adopt memory-mapped registers (MMR) to perform both configurations (*i* and *ii*).

4.2.3 NM4SMP Hardware

Figure 4.3 shows a more detailed view of the implemented NM4SMP hardware architecture that augments each core of the SMP. The additional logic is located near the interface between the core and L1-D cache to capture memory accesses. The NM4SMP hardware requires three components (Local Scratchpad Memory (`SPM`), Control logic (`Listener Cntrl` and `Notifier Cntrl`), Cache Address Blocker (`CAB`)) that are described in the next subsections.

Local Scratchpad Memory (SPM)

The SPM stores the FN matrix, each line (row) represents a consumer actor a, and each bit of the line (column) represents the data availability of its input FIFO i for the consumer actor (firing rule). As SPM is a scratchpad, it has an access latency of one cycle [16]. The SPM size is fixed at design-time. The studied benchmarks that represent various dataflow applications do not require more than $SMP_{size} = 2KB$ and $|FN| = 128$, which is a significant amount of maximum communicating pairs (producer and consumer actors). This represents 128 possible simultaneous actors for each core in a SPM size of 2KB and each actor is communicating with 128 actors. In the case of static dataflow graphs, we calculate the number of actors after graph flattening at compile time. If this number exceeds the size of SPM, the exceeded communication will be handled purely in the software. For reconfigurable dataflow application, the token rate can increase at runtime and hence, the number of actors might exceed the SPM size. In this situation, the rest of the actors are handled through software synchronization.

Control

The Control unit includes the following components: Listener, Notifier, Base Address Register (BAR), buffer, and a 128-bit AND gate. The BAR is configured at booting time by the middleware which is the base address of the FN matrix (48-bit size virtual address). All accesses to the FN performed by the core are in range of BAR. Therefore the Control unit can monitor such FN accesses and implement its logic. The key idea is that based on the address range stored in BAR Figure 4.2**(a)**, Notifier monitors the address bus between the core and L1-D cache and can detect when a write operation happens to FN matrix. The read and write operation is determined by a signal coming from the core that indicates whether the memory access is a load or store.

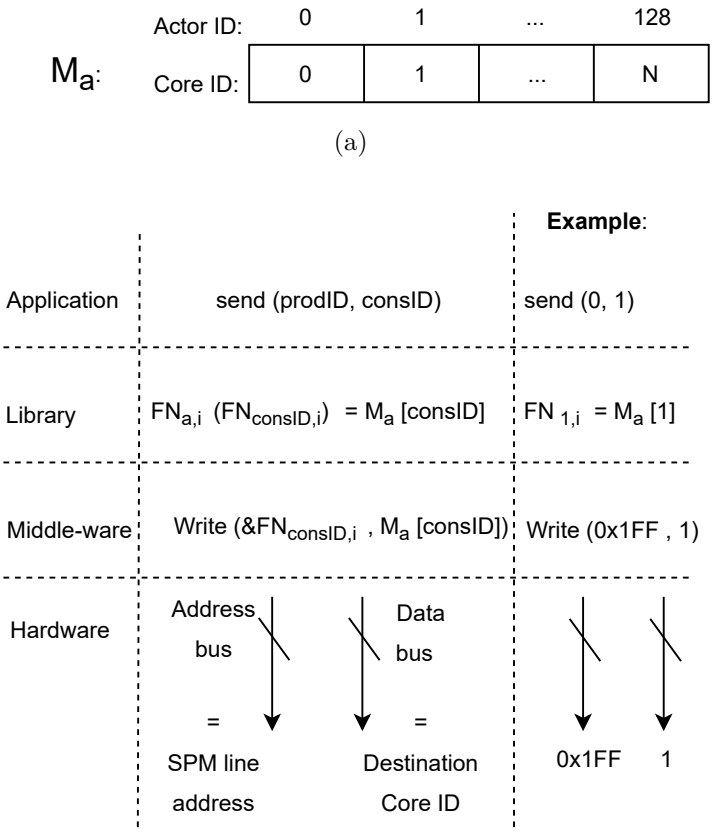


Figure 4.4 – (a) Array of mapping information. (b) Flow of updating SPM by capturing the Write and sending Notification.

Cache Address Blocker (CAB)

The CAB is an AND gate enabled by the Control logic. The CAB goal is to prevent that the read/write accesses to the FN reach to the lower levels of cache hierarchy (e.g., L2 cache). Memory operations with addresses in the range of BAR inevitably enter in L1-D that can generate a cache miss, triggering the coherence protocol. The NM4SMP logic detects such addresses and disables the interface between the L1-D and the next cache level by enabling the 2-input AND gate depicted in Figure 4.2**(e)**. One of the AND inputs is the valid signal from the L1-D, and the other is the signal coming from Control logic of NM4SMP. By setting the signal of CAB to 0, the output becomes 0, making the next cache level not be acknowledged by the cache miss message.

Figure 4.4 provides an example of the flow of sending Notification message in implementation stack of NM4SMP. In the mapping scenario depicted in Figure 4.1 when the

producer actor 0 finishes the execution, it writes data into the i th input FIFO of the consumer actor 1. Then the `send(0, 1)` is called. At the library level inside the send function, based on the mapping information provided in mapping array M_a (shown in Figure 4.4), the destination core ID 1 which hosts the consumer actor 1 is retrieved. Afterwards, this value will be written into the related flag $FN_{1,i}$. At hardware level, this flag address ($0x1FF$) which is in range of the BAR address (Figure **4.2(c)**) and its related data which is destination core ID 1 are captured by the NM4SMP logic. As discussed previously, this address is mapped to the SPM and hence, Notifier sends this address as a *Notification*, implemented as a specific NoC packet, to the Listener Figure 4.2 **(b)** of the destination core 1 that hosts the consumer actor 1.

The *Notification* packet travels through the NoC and is stored in the buffer on the consumer side **(c)**. The Listener handles this packet and sets on SPM set to 1 the value related to this address (value of the flag $FN_{1,i}$) **(d)**. In parallel, Listener detects (in the same way as Notifier) when the consumer core performs a read operation on FN. By performing a 128-bit AND operation **(e)**, the Listener can know when the firing rule of the consumer actor a is satisfied and reports to the core **(f)**. The subsection 4.2.4 presents further details, explaining the FSMs of Listener and Notifier.

4.2.4 NM4SMP Workflow

The workflow includes four phases: initialization, notifying, listening, and reporting.

Initialization Phase

This phase is performed each time the *initialize* is called. Its goal is to initialize the SPM with zero in firing rule of a ($FN_{a,i}$). The interface between the middleware and hardware module is implemented through a memory-mapped register (MMR), which fires `init` state in all FSMs depicted in Figure 4.5. After initialization, the NM4SMP control starts notifying, listening, and reporting phases.

Notifying Phase

The notifying phase, depicted in Figure 4.5(a), is triggered by the *send* function and implemented by the Notifier module. As previously explained and shown in Figure 4.4(b), Notifier detects the write access to the $FN_{a,i}$, then it sends a packet to the consumer core. This process is implemented by two FSM states depicted in Figure 4.5(a):

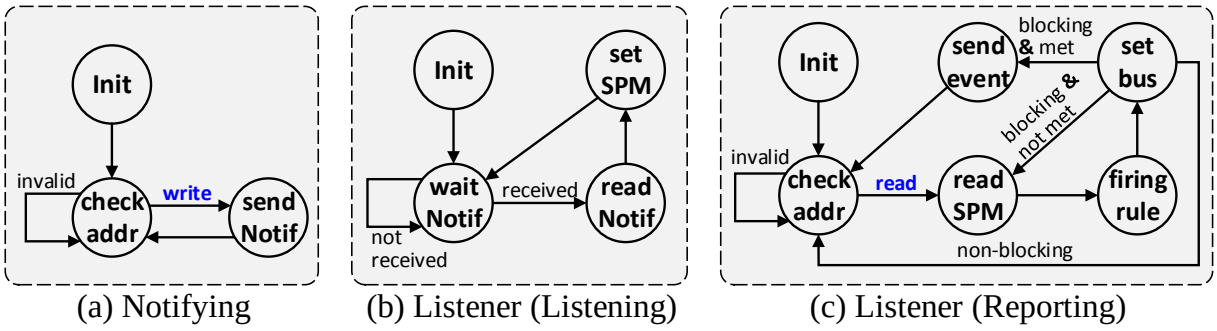


Figure 4.5 – NM4SMP Finite State Machines (FSM)

— `check addr`: Using the base address initialized in BAR, the Notifier detects address and captures its related data from the address and data buses respectively. When the core performs write operations, Notifier captures address and data buses. As the address bus is synchronized with the value of the data bus (both appear in the same cycle), the extraction of address ($FN_{a,i}$) and data (destination core ID) is performed within one clock cycle. If the address is in the range of FN, Notifier sends this address to the destination core whose ID is captured from the data bus as a *Notification* packet. Moreover, in case the address is in the range of BAR, this state also sets the CAB signal to 0 to block the cache miss access from L1-D to the next level, and FSM switches to `send Notif` state.

— `send Notif`: The Notifier sends a *Notification* packet to the Listener at the consumer core via its Network Interface (NI). The content of this packet is the address of the $FN_{a,i}$ flag. This process takes one clock cycle since NI only assembles and sends the NoC packet. Therefore, the goal of NM4SMP in this state is to pass minimal information to NI. The minimal information is the tuple $\{a, i\}$ and the destination core address. After this state, the Notifier FSM returns to the `check addr` state.

Listening Phase

In the listening phase, the Listener module receives the notification packet and updates the SPM. This process is implemented by three FSM states depicted in Figure 4.5(b):

— `wait Notif`: In this state, the Listener of consumer core waits for a notification packet from NI. When it receives the packet, FSM switches to `read Notif`.

— `read Notif`: the packet is read from NI, and the address of $FN_{a,i}$ is extracted. In sequence, the FSM switches to `set SPM`.

— `set SPM`: The firing rule located at $FN_{a,i}$ is set to 1 and the *update SPM signal* is set. Then FSM returns to the `wait Notif` state.

Reporting Phase

When a *receive* is called, and the related firing rule is not met, the actor's execution can be suspended. Differently than in spinning-lock, this action reduces the core's switching activity and helps to save energy. When the firing rule becomes enabled, it is necessary to report this event to the core to allow the actor to resume its execution. The report phase inside Listener has this role. When the Listener detects a read access to FN by monitoring the address bus, it checks the firing rule of the actor a, making a simple AND among all flags i of FN. If the output of the AND is 0, the rule is not satisfied, and the actor execution is blocked. Otherwise, the firing rule is met, and the actor execution continues. This rule checking is only performed once (whenever a *Notification* packet is received and SPM is updated) instead of constant spinning on it. While this behavior models a blocking *receive*, the support of non-blocking also was implemented. The only difference in the non-blocking case is that the core does not enter in the idle state when the firing conditions are not satisfied (the Listener put 0 on data bus and *receive* function returns zero). The actor is blocked by the thread since the *receive* returned zero, but the thread can still executing the other actors.

The reporting phase is composed of the following FSM states depicted in Figure 4.5(c):

— `check addr`: Listener checks whether a read happens to FN. When the read is detected, the CAB is set to 0 (preventing cache misses), and the FSM goes to `read SPM` meanwhile core should wait for the response (setting data bus as 0 by NM4SMP).

— `read SPM`: based on the read address, the Listener reads the line of FN that hosts the firing rule for actor a. The Listener calculates the firing rule for a using the 128-bit AND gate (128 bits is due to the maximum number of i). The result is 1 only if each flag i for actor a is 1, so the firing rule is met.

— `set bus`: Listener sets the firing rule status value (0 or 1) on the core data bus. If the *receive* is non-blocking, the FSM sets the bus as 0 (if the firing rule is not satisfied) or 1 (if the firing rule is satisfied) and returns to `check addr` to detect the next possible reads. If the *receive* is blocking, the core stays waiting and the FSM can take two directions: (1) if the firing rule is satisfied, the FSM sets the bus as 1 and the core needs to be awakened, thus, the FSM goes to `send event`;

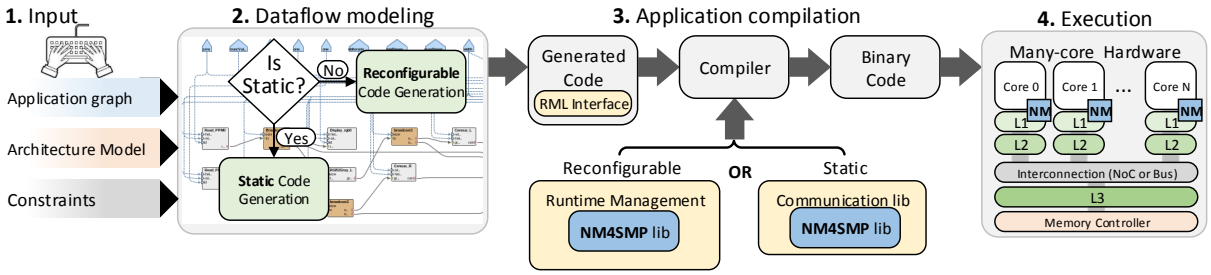


Figure 4.6 – Flowchart of the proposed framework to integrate NM4SMP co-design in the process of dataflow modeling, compilation, and execution.

(2) if the firing rule is not satisfied, the FSM sets the bus as 0 and it stays in the current state until the *update SPM signal* becomes set (by the listening phase) to switch to the `read SPM`.

— `send event`: Listener sends an event to the core so that the waiting core is awakened. Then, FSM state changes to `check addr`.

4.3 Notifying Memory Framework for PiSDF Dataflow Applications

This section presents the second contribution of this chapter: a framework to integrate the NM4SMP co-design in the process of dataflow modeling, code generation, and compilation of static and reconfigurable dataflow applications. Figure 4.6 presents the framework's flow. We divide it into steps which are explained in the following:

Step 1 - Input: the goal of this step is to gather all necessary inputs to model the dataflow application. The user must provide the following set of information: (*i*) the application's graph, containing the actors as vertices and its respective communication as the edges; (*ii*) the architecture model, detailing hardware properties of the target platform, as the number of cores, memory size, memory bandwidth; and (*iii*) the constraints of the application, as period, deadline, execution time, and minimal communicating bandwidth required by each edge.

The type and format of such information rely upon the dataflow modeling methodology used in the next step.

Step 2 - Code generation: the goal of this step is to generate the application code according to the input provided in the previous step. The output of this step is the dataflow

code generated in a given language. Rapid prototyping frameworks, as PREESM [110], are specialized in modeling dataflow application. As discussed earlier in Section 2.4, PREESM is a state-of-the-art framework for modeling and generating both static and reconfigurable dataflow applications for multi/many-core systems. PREESM allows graphically modeling the application by creating actors and their respective communicating edges, specifying the architecture model, and defining the application's constraints following a GUI based on Eclipse. We use PREESM to model the dataflow applications.

After the user models a dataflow application, the dataflow modeling tool can generate the code based on a target language (usually C). The code generation process requires an important step, which is to identify if the application is modeled following a static or reconfigurable approach. In the static approach, the application is already generated with the optimal mapping and scheduling since the data token production rate is deterministic.

For reconfigurable dataflow applications, a Runtime Management Layer (RML) can dynamically map and schedule the actors at runtime, according to the input workload characteristics (e.g., image frame size). In such a case, the dataflow modeling tool must generate an interface that becomes part of the generated application and makes the bridge between the application and the RML (see the yellow filled part of Figure 4.6). This interface has two purposes: (*i*) provide to the RML the constraints of application and properties of the target platform, (*ii*) allow the RML to know the address of the function of each actor, allowing the RML to start execution of them according to its runtime scheduling and mapping decisions. In PREESM, such interface is automatically generated when it is set to generate a reconfigurable application [110, 62].

We use SPiDER as RML [62]. The SPiDER RML library is modified to exploit the NM4SMP hardware module. Subsection 4.3.1 explains the contributions made inside the RML to support NM4SMP.

Step 3 - Application compilation: the goal of this step is to compile the dataflow application having as input the generated code and producing as output the binary code ready to be executed on the target platform. During the compilation step, the generated code (including the application's code and RML interface) is linked to the libraries that implement RML.

Step 4 - Execution: the binary generated in previous step can be executed on the simulated target multi-core architecture using full system simulator in which NM4SMP module is implemented, here we use Sniper multi-core simulator.

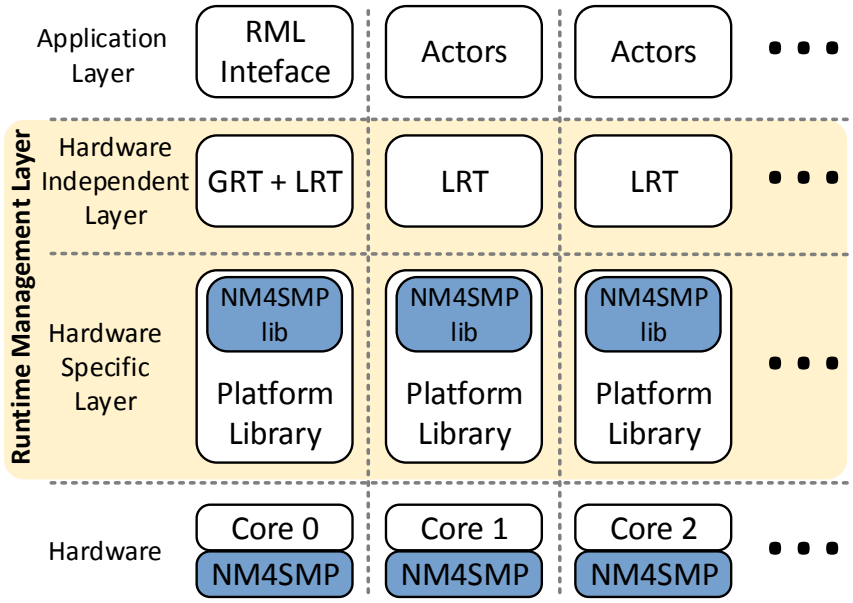


Figure 4.7 – SPiDER Runtime Management Layer (RML) structure. The blue rectangles are new modules added in this work.

4.3.1 SPiDER's Runtime Management Layer

The NM4SMP library depicted in Figure 4.6 is added a state-of-the-art RML called SPiDER [62]. We provide a detailed explanation of SPiDER in Section 2.4.2. It manages reconfigurable applications through hierarchical runtime management composed of one global manager (GRT) and n local managers (LRTs) where n is the number of threads running the application. GRT is responsible for managing the application graph, performing mapping and scheduling the actors at runtime and sending the job schedules to the LRTs at the beginning of each iteration. LRTs are responsible for executing the actors provided by GRT.

Figure 2.6 shows the SPiDER runtime management layer structure. The first layer is the application layer, having the RML interface and a set of actors to execute. The RML interface provides GRT the information related to architecture and application properties such as graph, functions of the actors and number of the threads.

At the bottom of the application's layer is the RML layer (filled yellow part), divided between Hardware Independent Layer (HIL) and Hardware Specific Layer (HSL). HIL implements hierarchical management with one GRT and several LRTs. HSL implements architectural dependent libraries including communication library (NM4SMP library) according to the target platform.

Finally, the last layer of Figure 2.6 comprises the Hardware, where are the implemented NM4SMP hardware module.

The NM4SMP library is implemented within HSL by replacing the traditional semaphore-based synchronization adopted in SPiDER with the proposed approach. A fundamental difference between static and reconfigurable applications, is that in reconfigurable case the actors do not call the *send* and *receive* functions directly, instead, the LRT abstract those functions to the actors by handling data communication and synchronization among the actors. At the beginning of each application iteration, GRT, based on the decided scheduling and mapping, provides to each LRT a list of actors (scheduled jobs) to execute. Each LRT, after it finishes the execution of an actor (running its function), sends a notification (called job notification) to the other LRTs hosting other actors who are waiting for the data produced by this actor. Thus, the LRTs communicate among them to know when the firing rule of a given actor were satisfied, allowing the LRT to start the execution of that actor. Such communication among LRTs plays an important role in the communication overhead of the application (up to 40% in our studies), and it can be exploited to reduce the communication penalties by implementing it with NM4SMP.

4.4 Results

This section is organized as follows: subsection 4.4.1 details the experimental setup. The next two subsections evaluate performance (execution time, cache impact, energy consumption, and scalability). Two different scenarios are addressed: subsection 4.4.2 evaluates SDF applications *without* using SPiDER. Such an evaluation is relevant because SDF applications call the *send*/*receive* functions directly (without the RML), which makes the evaluation of NM4SMP clearer since there is no RML overhead. Subsection 4.4.3 *includes* RML in the dataflow applications generation, following the framework steps presented in section 4.3. Such results are sub-divided in SDF applications generated with RML and the evaluation of a fully reconfigurable application (PiSDF). To conclude, the subsection 4.4.4 details the hardware complexity imposed by NM4SMP.

4.4.1 Experimental Setup

Dataflow Benchmarks

We evaluate four dataflow applications implemented as SDF graph

Table 4.1 – Benchmark setup. **RLT** / **RLP**: Reinforcement Learning Training / Prediction phases. **SDF**: Synchronous Dataflow. **PiSDF**: Parametererizable Dataflow

Application	Actors	FIFOs	FIFOs size	Graph
RLT	61	2111	42.7 KB	SDF
RLP	10	282	7.7 KB	SDF
Stabilization	30	607	921 KB	SDF
Stereo	36	811	29.09 MB	SDF
Sobel	5	43	681 KB	PiSDF

- Reinforcement Learning application Training algorithm (RLT)
- Reinforcement Learning application Prediction algorithm (RLP)
- *Stabilization* (a filter to compensate the movements of a video recorded with a shaky camera)
- *Stereo* (a computer vision application that processes a pair of images to produce a disparity map corresponding to the depth of the captured scene)
- *Sobel* filter as a reconfigurable application and modeled as a Parameterized and Interfaced Synchronous DataFlow (PiSDF)

Table 4.4.1 presents the features of our benchmarks. The dataflow applications come from PREESM's application repository [1] and are specified using the PREESM framework [110]. While it is expected that applications with a high communication rate will benefit more from NM4SMP, the applications set herein adopted is interesting because it comes with a mixed profile of computation, memory usage, and communication, which allows evaluating different aspects of the NM4SMP approach.

Hardware Setup

Table 4.2 summarizes the two SMP configurations adopted in our experiments. They are by default available in Sniper distribution. The first one, similar to the experimental setup in Chapter 3, is Xeon X550 Gainstown, which represents a mainstream and powerful SMP. The other one is Atom Silvermont, which represents a low power SMP. Each one of these systems was evaluated with NM4SMP and without (baseline) using Linux-based semaphore [27] (kernel 5.4.0-84) as a synchronization technique.

1. https://github.com/PREESM/PREESM-apps

Table 4.2 – Hardware simulation settings.

Feature	**System 1: Xeon-based SMP**	**System 2: Atom-based SMP**
Processor type # Cores	Intel Xeon X5550 Gainestown 16 core @ 2.66 GHz	Intel ATOM Silvermont 16 core @ 2.4 GHz
L1-I Cache	32KB - 8way - 1ns tag lat. 4ns Data lat. - LRU	32KB - 8way - 1ns tag lat. 4ns Data lat. - LRU
L1-D Cache	32KB - 8way - 1ns tag lat. 4ns Data lat. - LRU	24KB - 6way - 1ns tag lat. 4ns Data lat. - LRU
L2 Cache	256KB (private) - 8way 3ns tag lat. - 8ns Data lat. - LRU	1MB (shared by 2 cores) - 16way 3ns tag lat. - 12ns Data lat. - LRU
L3 Cache	8MB (shared by 4 cores) - 16way 10ns tag lat. - 30ns Data lat. - LRU	No L3 cache

lat = latency; LRU = Least Recently Used.

4.4.2 SDF Applications Without RML

This subsection addresses the results for the static application modeled by SDF graph. The results focus on evaluating the potential of NM4SMP to reduce the synchronization overhead, addressing metrics of execution time, cache access/miss, and energy consumption.

Synchronization Time

Figure 4.8 presents the normalized execution time (performance) evaluation, divided in communication, computation, and memory time. Each application has three bars: baseline, *baseline-64*, and NM4SMP.

The *baseline-64* bars represent a study that evaluates the impact of a 64 KB L1 cache. Such a study was performed because the *baseline* and NM4SMP platforms use an L1-D cache size of 32KB for Xeon and 24KB for Atom. Details of the hardware settings of *baseline* architectures are represented in Table 4.2. By adding NM4SMP, the cache size increases due to the presence of SPM. A reasonable argument is that simply using a bigger L1 cache would speedup applications. We thus wanted to measure the impact of such an increase by evaluating configurations with a bigger L1 cache. Due to the limitation of Sniper, the cache size must be a power of 2, therefore *baseline-64* configuration adopts the size of 64KB L1 cache for Xeon and 32KB L1 cache for Atom.

The results of Figure 4.8 show that *baseline-64* presents an insignificant impact on application execution time compared with the baseline system, with an average speedup of less than 1% for all applications. Hence, we decided to keep the configuration with an L1-D cache of 32 KB for Xeon and 24 KB for Atom as baseline configuration and compare

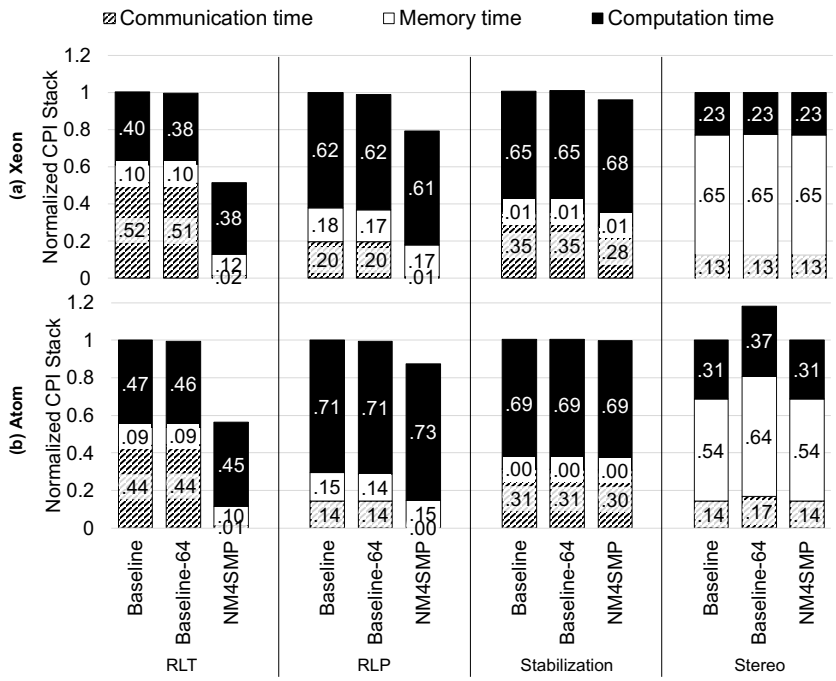


Figure 4.8 – Normalized CPI stack of static dataflow benchmarks for Xeon and Atom.

both with the same system plus the presence of NM4SMP. We argue that this leads to a fairer energy evaluation, enabling detecting the additional overhead of the NM4SMP module.

Comparing in Figure 4.8 the NM4SMP with baseline is possible to observe that, in overall, NM4SMP presented a average speedup of 1.23× considering both the results of Atom and Xeon. The performance results addressing Xeon depicted in Figure 4.8(a) shows that NM4SMP presents a speedup up to 1.96× (for RLT) compared with baseline, with an average of 1.31× among all applications. The performance speedup was insignificant for Stereo (1×), since we observed that Stereo is bounded by memory operations (e.g., *memcpy()*) and not communication. On the other side, RLT is communication bounded, allowing it to benefit from NM4SMP.

The results for Atom (Figure 4.8(b)) are similar to Xeon. The best speedup was achieved for RLT (1.78×), and Stereo also did not show any improvement. The average speedup among all applications is 1.22×. NM4MP presents a slightly lower speedup in Atom than Xeon, which is due to the smaller processor. In a slower processor, increasing the portion of computation time, the impact of memory access in execution time becomes less significant than the bigger cores. Therefore, NM4SMP has less opportunity to improve performance by saving memory access in the cache subsystem of Atom, which creates more

Table 4.3 – Performance and synchronization metrics.

Applications	RLT	RLP	Stabilization	Stereo
		Baseline		
Execution Time (ns)	3.15E+08	4.07E+08	3.49E+08	1.72E+09
Practical synchroniz. time (ns)	1.62E+08	8.02E+07	4.35E+07	2.15E+08
STR	1.05	0.24	0.53	0.14
		NM4SMP		
Execution Time (ns)	1.60E+08	3.26E+08	3.45E+08	1.72E+09
Practical synchroniz. time (ns)	2.52E+06	2.44E+06	3.35E+07	2.15E+08
STR	0.03	0.009	0.41	0.14
NSync	1.47E+06	9.68E+05	2.22E+04	4.17E+03
Avg Synch Speedup	**64.48**×	**32.93**×	**1.3**×	**1**×
SETC	**9.37**	**0.89**	**0.02**	<< **0.01**
Application Speedup	**1.97**×	**1.25**×	**1.03**×	**1**×
Instruction saved	**17.85%**	**4.51%**	**0.02%**	**0.00%**

RLT / RLP = Reinforcement Learning Training / Prediction phase;
Avg = Average; Synch = Synchronization;

DRAM accesses.

Note that the applications present a disparate speedup (as in the case of RLT and Stereo). To better understand this disparity and the impact caused by NM4SMP, we analyze the applications' behavior in Table 4.3 using Xeon results as a case study. The values of line *Avg Synch Speedup* show the speedups for synchronization. The value demonstrates the NM4SMP improvement compared to semaphores. In semaphores, the actors communication is managed by *sem_post()* and *sem_wait()* operations. Analyzing in Figure 4.8, it is possible to conclude that applications spend up to 52% of their execution time communicating (RLT). This time includes two types: **(1)** waiting time for data to become available, which is computation dependent, **(2)** the cache memory subsystem overhead to ensure atomic operations on semaphore spinning-lock. From the application point-of-view, the time spent in *sem_wait()* function for catching the lock is wasted time. By replacing the semaphore implementation with NM4SMP, such overhead is minimized. This is more efficient than multiple atomic operations required by semaphores involving the cache hierarchy.

To analyze the synchronization precisely, we introduce two parameters. First, we introduce Synchronization Time Ratio (*STR*) as the synchronization time of the whole application execution. *STR* is defined by Equ. 4.1, where N is the number of cores, T_{Sync}

is the number of cycles spent on *send* and *receive* functions, and T_{Exec} is the number of cycles spent on the actor's function (its processing part).

$$STR = \frac{1}{N} \frac{\sum_{i=1}^{N} T_{Sync}(i)}{\sum_{i=1}^{N} T_{Exec}(i)} \tag{4.1}$$

STR represents quantitatively the synchronization overhead in dataflow applications. Table 4.3 shows the STR and the synchronization speedup for all the applications. The bigger STR, the higher chance to improve the synchronization by NM4SMP. Therefore, NM4SMP is not able to accelerate the Stereo application as it presents a high memory operation time. Performance improvement of NM4SMP depends on how much the application is synchronization bound. As the second parameter, we introduce a metric called Synchronization Events per Thousand Cycle (*SETC*), which is defined by Equ. 4.2, where N is the number of cores, N_{Sync} is the number of *send* and *receive* function calls, and T_{Exec} is the same as Equ 4.1.

$$SETC = \frac{1000}{N} \frac{\sum_{i=1}^{N} N_{Sync}(i)}{\sum_{i=1}^{N} T_{Exec}(i)} \tag{4.2}$$

This equation shows the frequency of synchronization calls in an application. Suppose an application that relies heavily on *send*/*receive* calls (i.e., high *SETC*). In that case, it means that communication between actors is not primarily determined by the actor cost function. The RLT and RLP applications depict this behavior, where the portion of computation is not considerable (see Figure 4.8), and the waiting time of *receive* function is from type (2) (cache hierarchy overhead). For applications presenting a high synchronization frequency and low computation, the *SETC* factor is bigger. On the contrary, in dataflow applications modeled by PiSDF where the generation of data tokens is limited by actor at execution time, such as the Stereo, the *SETC* is lower. This makes Stereo belong to type (1) (data dependency due to heavy memory transactions). Hence, contrary to the learning applications, Stereo has a very low *SETC* and can not benefit from NM4SMP. As shown in Table 4.3, the trend in execution time speedup and *SETC* is correlated. In other words, the lower *SETC*, the lower frequency of synchronization and the less computation bounded synchronization. Consequently, NM4SMP can improve a synchronization better when an application presents a high *SETC*.

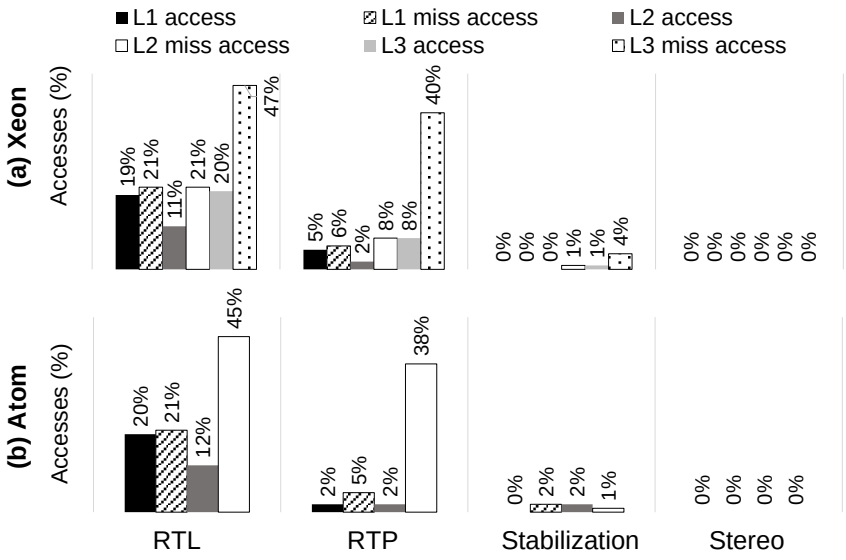


Figure 4.9 – Cache access reduction using NM4SMP for Xeon (a) and Atom (b) processors.

Cache Access and Miss

Figure 4.9 shows the percentage of cache accesses and misses saved by using NM4SMP compared to the baseline for Xeon and Atom processors. As expected, the most communication-intensive applications receive more benefits from the NM4SMP. Specifically, RLT and RLP have shown the best results, as can be observed in Table 4.3.

In general, the improvements of all applications between Xeon and Atom architecture is similar while using NM4SMP. Stabilization and Stereo do not present a significant improvement in cache access/miss. As already observed, these applications are not synchronization intensive (low *SETC*).

Such results corroborates our hypothesis that using an optimized module to speed-up constant locks checks contributes significantly to reduce the pollution and pressure of cache memories of dataflow applications. Such gains, besides impacting positively the execution time (as previously addressed by Figure 4.8), also play an important role to reduce energy consumption, as will be presented in the next subsection.

Energy Consumption Evaluation

This section studies the impact of NM4SMP on the energy consumption of the whole system. To perform such an evaluation, we use McPAT [82] integrated in Sniper to measure the energy consumption of SMP architecture. The energy model of NM4SMP was profiled

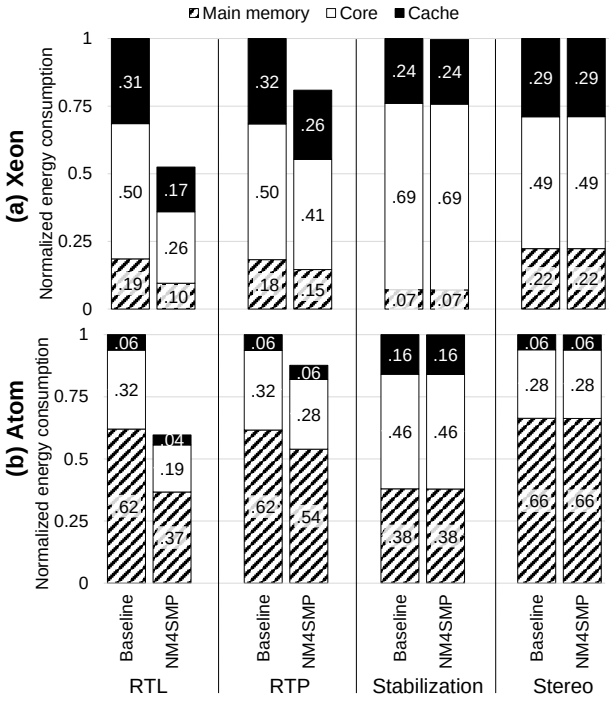


Figure 4.10 – Normalized energy consumption stack for Xeon (a) and for Atom (b) processors.

and added to the McPAT model. The energy model of NM4SMP is comprised of static and dynamic models based on CACTI [101], which is considered in the following as a reference for the power consumption of different memory and caches implementations.

Among the sub-components of NM4SMP, the SPM, configured with the worst-case size of 2KB SRAM, determines the energy consumption of NM4SMP. The static energy of SPM is measured by feeding its configuration to CACTI. To model the dynamic energy of NM4SMP, we use counters inside our Notifiers and Listeners to measure the number of accesses to the SPM. We then set an energy model based on CACTI, which calculates the dynamic energy based on SPM accesses. Due to the Sniper limitations to model the NoC, NoC energy was not taken into account. This makes the evaluation herein presented more pessimistic since one expected feature of NM4SPM is to reduce the cache traffic over the NoC.

Figure 4.10 depicts the impact of NM4SMP on energy consumption. Both architectures present close energy reduction, 16.7% for Xeon and 13.2% for Atom. The overall energy reduction is 14.98% on average for both systems compared to the baseline. Such energy savings come thanks to two main factors: (*i*) NM4SMP decreases energy consumption of

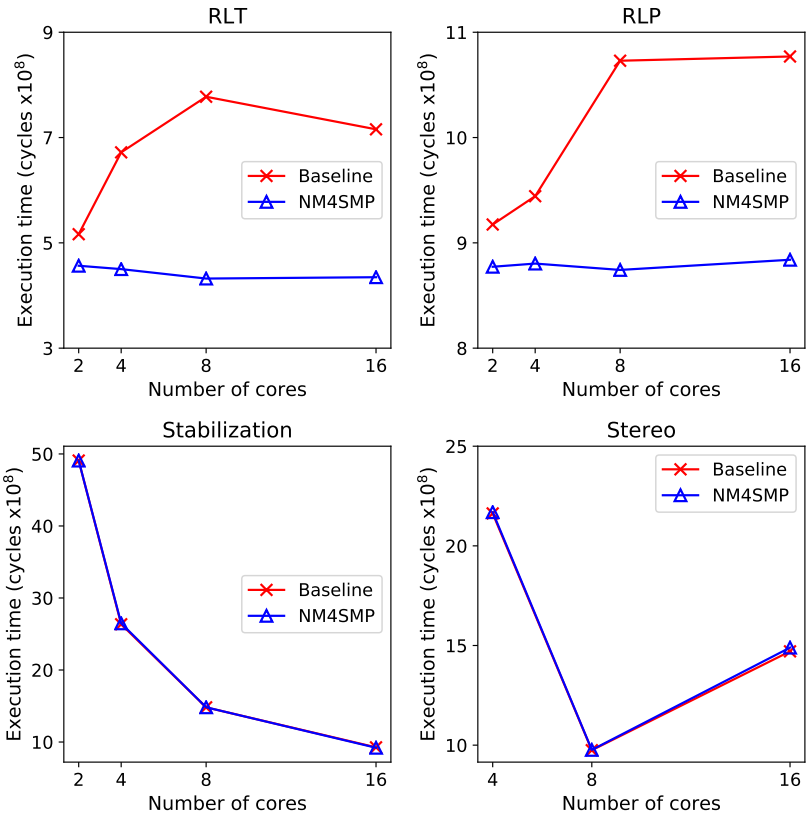


Figure 4.11 – Scalability of the static applications.

caches and main memory by preventing polling accesses and related misses in cache hierarchy; and (*ii*) the OS overheads of interruption, context saving/restoring, and semaphore execution is reduced since the actors are only triggered when all firing rules are met.

Scalability Evaluation

Figure 4.11 shows a scalability evaluation addressing the impact of NM4SMP in the execution time according to the number of cores. The increased number of cores makes PREESM split the application actors in order to make use of all the cores available. The results in Figure 4.11 can be explained in two parts: applications that benefit from NM4SMP (RLT and RLP) and applications that do not (Stabilization and Stereo).

While observing only the baseline results of RLT and RLP, it is possible to see that increasing the number of cores is not beneficial for the applications since its execution time increases. This happens because such applications were not designed to be fragmented in more than 2 or 4 cores. The PREESM git repository offers a limited number

of functional benchmarks that includes: RLT, RLP, Stabilization. The other benchmarks were not implemented in a parallel way to benefit from NM4SMP.

The interesting behavior is that when NM4SMP is employed, this increased core count penalty is significantly amortized, presenting a reduction up to -44% for RLT (16 cores) and 18% for RLP (8 cores) compared to the baseline. This improvement is achieved since the synchronization overhead imposed by increasing the number of cores (cache movements especially) is amortized by the presence of NM4SMP. On the other hand, applications Stabilization and Stereo (as already expected based on the previous results) do not benefit from NM4SMP due to their memory and computation bound profiles. The only difference between those applications is that Stabilization scales better than Stereo, as already corroborated in Chapter 3. Our study considers different application behaviors with various benchmarks to avoid biasing the evaluation. Some benchmarks are actually computation bound, some other are memory bound and the others are synchronization bound. Additionally, we did not reject any application which has no gain from our technique, like Stabilization.

4.4.3 Applications With RML

This subsection provides the performance results (execution time, memory access, and energy) to applications generated with RML. The results include SDF and PiSDF applications.

SDF Applications

The SDF applications considered in this experiment are the same as the evaluation without RML except for the Stereo, which was excluded because its graph is not compatible with the SPiDER's methodology to prepare and transform the graph during runtime actor's scheduling [62]. Figure 4.12 presents the results of normalized execution times in CPI stack for the SDF applications running with RML. We observe that NM4SMP contributes to reducing the communication time of the applications on Xeon and Atom processors, on average by 27.13% and 28.91% respectively. It results in an application execution time reduction of 17.1% and 14.3% for Xeon and Atom (average of 15.7% considering both).

Figure 4.13 represents the cache accesses of the applications, showing that NM4SMP contributes to reduce the LLC cache miss accesses for Xeon and Atom on average by 5%

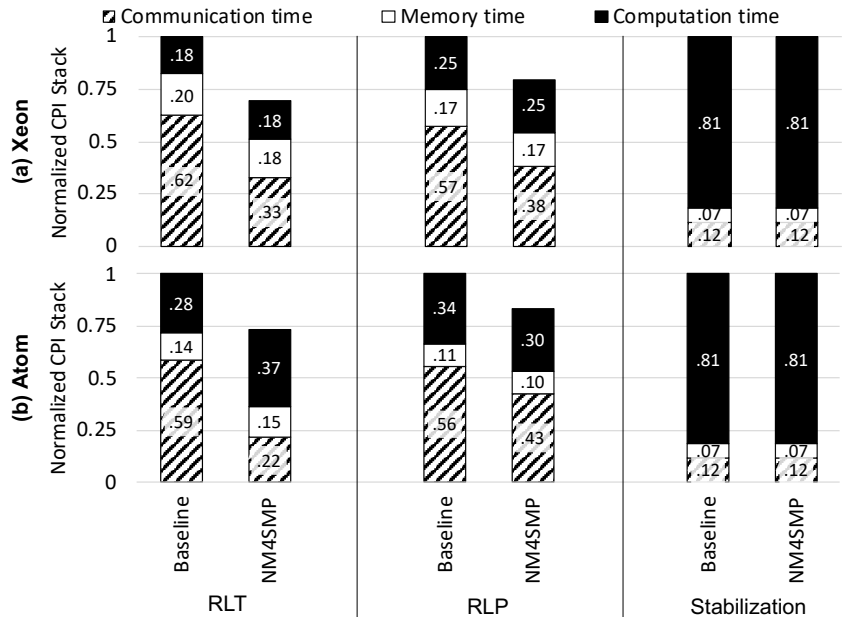


Figure 4.12 – CPI stack of the static application managed by RML.

and 3% respectively. As can be seen, NM4SMP contributes less in improving the performance by saving less cache accesses of the applications managed by RML compared with the same SDF applications but without RML (Figure 4.9). This is because of the intrinsic management overhead that the RML imposes in the application (e.g., communication among LRTs, GRTs, and LRTs, and other internal management functions).

Figure 4.14 illustrates the energy consumption reported by McPAT of Sniper on Xeon and Atom processors. Total energy consumption on Xeon and Atom is reduced on average by 13.21% and 12.04%, respectively. This is due to the reduction of the cache and main memory accesses and the increased idle time of the core, which is caused by NM4SMP that can signal the core only when all firing conditions were met.

We also study the scalability of the applications managed by RML with and without NM4SMP. Figure 4.15 illustrates the result of the execution time of the applications with RML running on different numbers of cores. As previously observed in the evaluation without RML, the applications RLT and RLP do not scale well. The same pattern remains in the case with RML. The adoption of NM4SMP has reduced the execution time, reaching -26.3% for RLT in 32 cores and -17.1% for RLP in 8 cores. However, due to the not scalable application design, the curve follows the same trend as the plot without NM4SMP. Also, note that this reduction is lower than in Figure 4.11. This happens because the RML needs

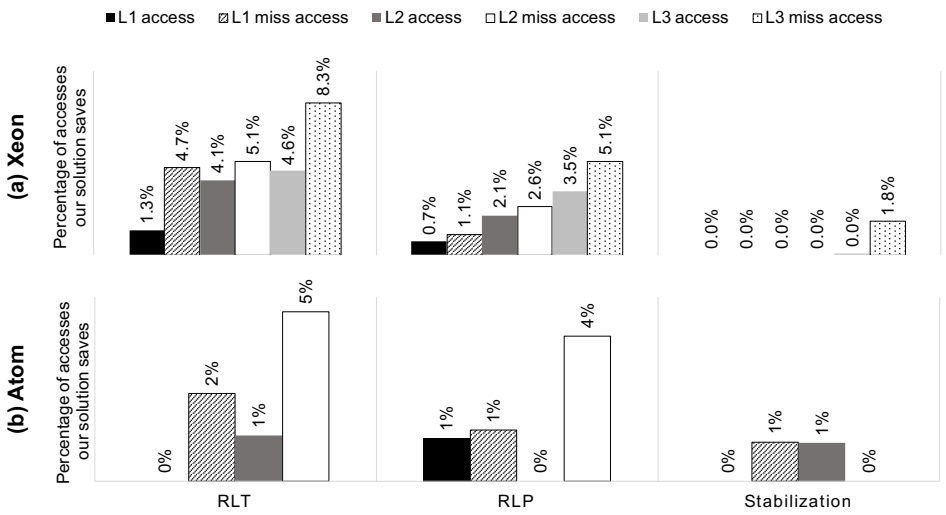


Figure 4.13 – Cache access reduction for static application managed by Spider

to implement more management communications between managers (LRTs and GRTs), which increases performance penalties.

On the other side, Stabilization is optimized to exploit an increased number of cores. In such a case, we observe no practical difference in using the NM4SMP since the difference of the mean of squared error between baseline and NM4SMP is negligible.

PiSDF Application with RML

This subsection evaluates the performance results of one PiSDF application, Sobel filter, running on Xeon and Atom processors. Figure 4.16 shows the normalized CPI stack of Sobel execution time. As can be seen, the behavior of the application on both processors is similar. The majority of communication overhead is removed by using the NM4SMP solution. Relative to the baseline system, Sobel achieved 17% speed up for both Xeon and Atom processors and a reduction, on average, of 14% cache access/miss.

Figure 4.17 presents the impact of NM4SMP on energy consumption. Both architectures present similar energy reduction, 13.3% for Xeon and 13.16% for Atom. The overall energy reduction is 13.25% on average for both systems compared to the baseline.

4.4.4 NM4SMP Hardware Complexity

The area complexity of NM4SMP can be evaluated analytically. It is mainly due to the introduction of an SPM attached to each core since the additional logic is limited to the

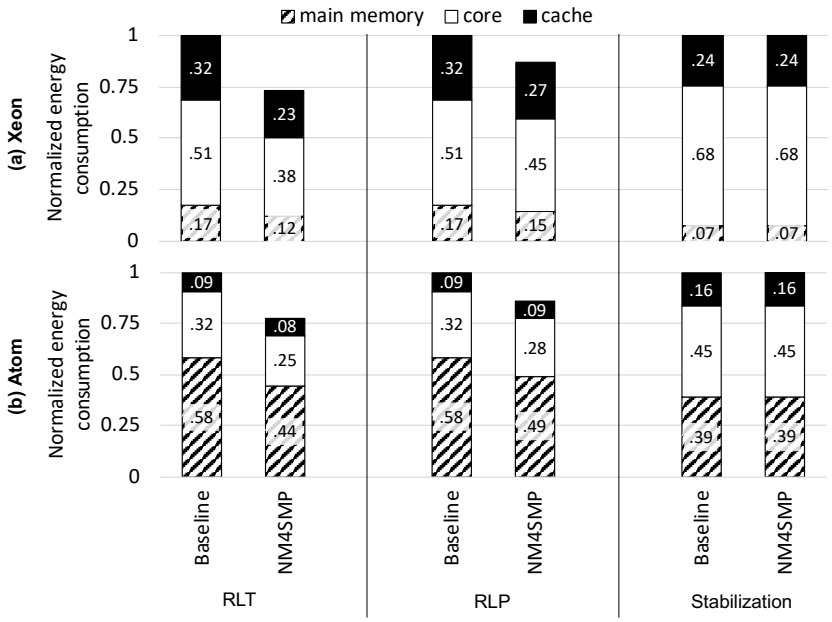


Figure 4.14 – Normalized Energy stack of the static application managed by RML.

controller part with FSM showed in Figure 4.5. The controller includes one comparator for the opcode, two comparators for address validation of FIFO flags, a 48-bit Base Address Register and few additional logic gates. According to [106], the area complexity of a 32-bit comparator is $218A_G$+$93A_N$ where A_G and A_N are the area of basic gate and Not gate, respectively. This area is less than 0.5% of the memory part of the L1-D cache (32KB) area, which is negligible.

For SPM area consumption, we consider a maximum size of 2KB, which supports 255 actors with up to 128 firing rules for each actor. This is an oversized setting as none of our benchmarks requires more than 64 actors. This SPM size (2KB) has an area consumption representing less than 1.01% of the L1-D cache size (32KB), which also requires a tag array and additional resources for implementing the cache coherency protocol.

For the power consumption evaluation, we focused on SPM since it dominates the complexity of NM4SMP. We use TSV40nm technology library modeled in CACTI to extract power figures. Figure 4.18 presents the total power (static + dynamic power) for NM4SMP considering different SPM sizes. As can be observed, the power follows a linear increase. To put in perspective such values, if we compare the SPM power assuming the higher SPM size in the plot (2KB) with the power required by an L1-D cache of 32KB (5700 mW), the SPM of NM4SMP represents just 82×10^{-6}% of the L1-D cache power. By

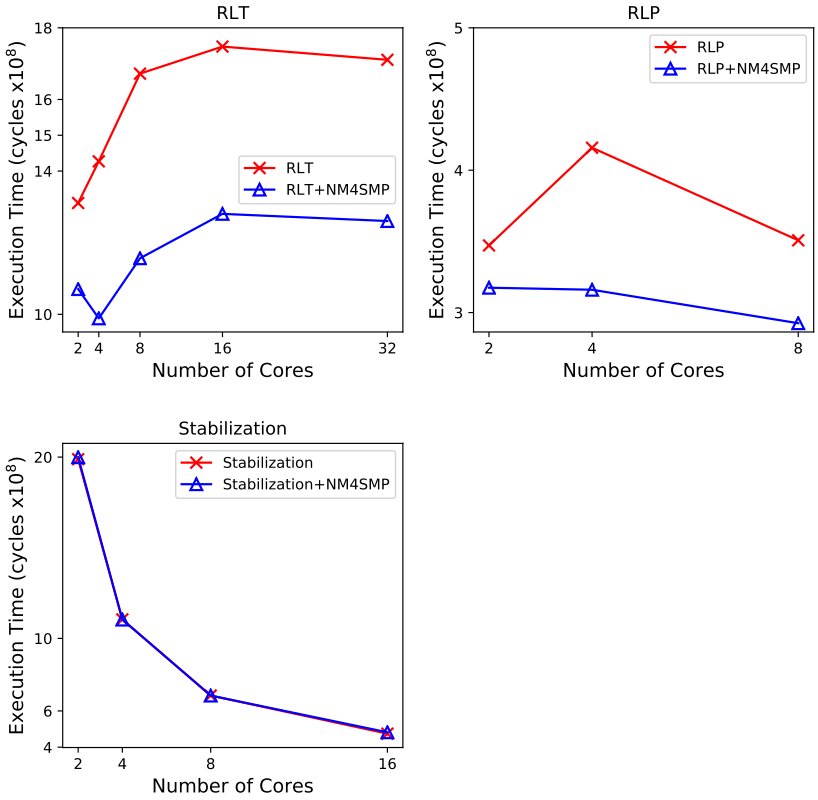


Figure 4.15 – Scalability of the static application managed by RML

observing such results, it is possible to conclude that NM4SMP presents a light hardware complexity, especially when compared to the benefits that it can bring in execution time and energy consumption.

4.5 Final Remarks

This chapter presented a near cache memory implementation of a module called NM4SMP optimized for dataflow applications. A HW/SW co-design implementation is presented which addressing library, middleware, hardware levels and presenting a framework that addresses how to include the NM4SMP in the process of dataflow modeling, code generation, and compilation, assuming static and reconfigurable applications.

Results corroborate our hypothesis that a specialized module, aware of dataflow behavior, can act not only to improve the synchronization of shared data but also to boost the application performance by considering the management of dataflow firing rules within

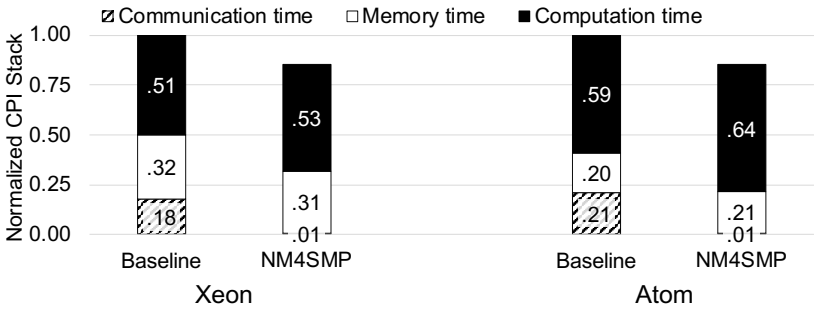


Figure 4.16 – Normalized CPI stack of Sobel for Xeon and Atom.

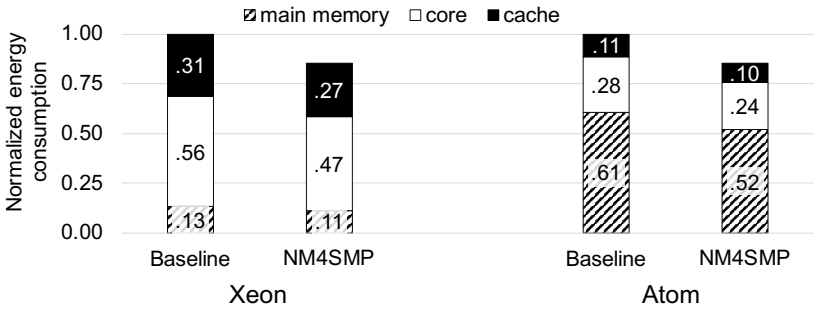


Figure 4.17 – Normalized energy stack of Sobel for Xeon and Atom.

its logic. The combination of synchronization at hardware with the dataflow firing rule awareness leads to important gains in application execution time and energy consumption. Results show an average speedup of 1.23× and an average energy saving of 14.98%, assuming two Intel SMP architectures and four applications. As expected, the application with a higher communication rate benefits more from NM4SMP. The technique also shows similar improvements when integrated to RML, reducing the applications execution time on average 15.7%.

Our solution opens interesting perspectives: (1) for dataflow parallelization in the sense that limiting the synchronization impact allows decreasing the actor granularity so increase the parallelism, (2) for multiple applications sharing the same cache.

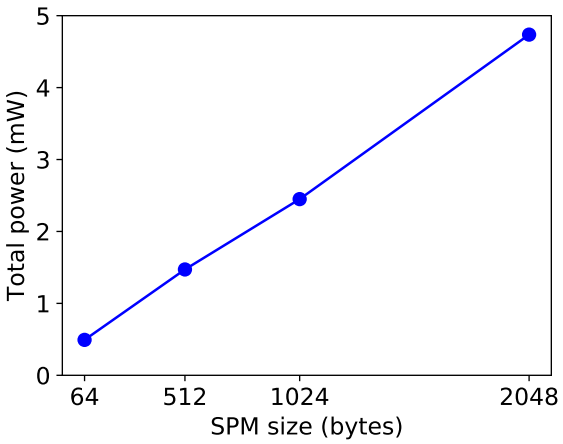


Figure 4.18 – Total power (static + dynamic) of NM4SMP module according to different Scratch Pad Memory (SPM) sizes.

Conclusion and Perspectives

The complexity of parallel applications in all industry fields, including digital signal processing and computer vision applications, has increasingly grown, and so their high-performance requirements. Among these applications, dataflow applications, thanks to explicit parallelism, can efficiently exploit SMPs, which are nowadays the most widely used high-performance multi-core processors. However, dataflow applications are not well adapted to this type of architecture as they specifically stress the memory hierarchy, limiting the performance. Thus, considering the gap between SMP architectures and Dataflow specific requirements, this thesis has addressed to revisit the hardware and software co-design of SMP multi-core architectures. The underlying topic of this thesis is a better understanding of the matching between SMP and dataflow applications. The main contributions are:

- In Chapter 3, we evaluate the cache behavior and workload characterization of dataflow applications. Afterward, we evaluate adopting existing dynamic memory management techniques, Copy-on-Write (CoW) and Non-Temporal-Memory (NTM) to improve dataflow application performance.
- A HW/SW co-design of a near-memory on-chip device to synchronize dataflow actors, called NM4SMP (Notifying Memory for SMP) to improve the synchronization on SMP, presented in Chapter 4.
- A Framework to integrate the NM4SMP co-design in the process of dataflow modeling and compilation of static and reconfigurable applications, detailed in Chapter 4.
- A detailed power and performance evaluation of dataflow applications (static and re-configurable) using NM4SMP for generic and low-power processors, reported in Chapter 4.

To perform dataflow workload characterization, in total, 37 different cache configurations (resulting in 213 simulations with three real applications) were adopted to evaluate variations in core count, L2/L3 sharing, and L2/L3 sizes.

The analysis based on different cache configurations achieves two main conclusions: (1) Bigger is not always better in core count, L2 sharing, and L2/L3 size since other aspects as efficient parallel workload division and computation/communication profile can prevent

the application from benefiting from more cache memory resources. (2) Private L2 and L3 shared among all cores provide the best speedup and L2/L3 cache miss figures for dataflow applications.

In the current context of larger number of cores and bigger caches, the first results of this thesis clearly show that: (i) achieving better performance is not just a matter of the higher number of resources; (ii) it is now a matter of efficiently using the available hardware and finding new hardware organization.

The advantages of employing the copy-on-write (CoW) and non-temporal memory transfer copies (NTM) are investigated in dataflow applications, and the results reveal that the techniques can improve execution time and save energy. NTM presents a modest reduction in execution time (up to 5.3%) and energy (up to 2.7%). CoW technique demonstrates important improvements for the applications with more significant memcpy transfers ($\geq$ 400KB) to overcome the overhead of this technique. The results show reductions of 15.8% in execution time and 21.8% in energy consumption. These dynamic memory transfer techniques are complementary to static state-of-the-art memory optimization approaches like [39] that reduce cache thrashing (NTM) and unnecessary data movements (CoW) among dataflow actors at runtime.

The proposed NM4SMP synchronization solution, with a HW/SW co-design implementation, by offloading the synchronization-related messages from cache-coherent memory hierarchy improves the synchronization of shared data. As an outcome, it boosts the application performance by managing dataflow firing rules within its logic. This firing-rule-aware synchronization results in an average speedup of 1.23$\times$ and an average energy saving of 14.98% for dataflow applications running on Intel SMP baseline architectures. The application with a higher communication rate benefits more from NM4SMP.

For the dataflow application managed by a Runtime Management Layer (RML), NM4SMP shows similar improvements that reduce execution time and energy consumption on average by 15.7% and 13.21%, respectively where using NM4SMP brings scalability joint with performance.

NM4SMP imposes negligible area and power consumption overhead which is mainly due to local SPM. In the worst case, local SPM with a size of 2KB has 1.01% of the L1-D cache size (32KB) area, and the controller area is 0.5% of the memory part of the L1-D cache (32KB) area. The majority of the power consumption of NM4SMP is consumed by the memory part, local SPM, which represents negligible power consumption in compared to L1-D cache (only 82×10^{-6}% of the L1-D cache power consumption). Finally, this thesis

has demonstrated that although SMPs provides high-performance parallelism, they do not present scalability, power and memory usage efficiency for dataflow application.

By introducing solutions in memory and more in synchronization, we have shown that their performance can significantly improve by exploiting the co-design hardware synchronization mechanism, NM4SMP.

Future Works

Based on the contributions of this thesis, several trails can be evaluated as future works. These perspectives can be categorized as long, mid and short-term perspectives.

Short-Term Perspectives

NM4SMP can help to reduce more the cache coherency burdens. At the same time that Notifier detects writing data by the producer and sending the *Notification*, it can push the cache to do *write through* down to the LLC to flush the written data of output FIFO into the shared level of cache with the consumer.

This idea avoids the cache miss in L2 of consumer core and the LLC when data is available but it is not yet fetched into the cache of consumer core.

Graph characterization is another suggested future work. Different dataflow graphs can be studied and characterized based on the features that impact the communication, including connection topology, edge count, actor granularity. Graphs with various communication degrees and broadcasting and multi-casting features require a different amount of synchronization and hence, benefit differently from the NM4SMP solution. Characterization of the graph can help finding the critical factor that has a prominent role in efficiently exploiting our proposed solution. With the help of this factor we can check whether the application has the capacity of exploiting NM4SMP or not. Therefore, this factor can be considered in other graph-based programming models rather than dataflow.

Adapting the NoC design to prioritize the notification packets throughout the interconnection network is another case study that can be done for the system using the NM4SMP synchronization solution. For instance, a bit flag in the header flit of the packets can be indicated to show whether the packet is a notification or not. Regarding this bit, the packet can pass through a specific virtual channel, a dedicated path for notification packets.

Mid or Long-Term Perspectives

NM4SMP solution can be evaluated for other memory architectures such as distributed and hybrid memory systems.

Our synchronization solution is not limited to general purpose architectures and any platform running dataflow application can advantage it. For instance, NM4SMP can be implemented in specific dataflow architecture or heterogeneous architectures with heterogeneous computing units (CPU, GPU, TPU and FPGA) and various memory bandwidths.

Other dataflow MoCs can make use of the proposed solution. For instance, DPN (Data Process Network), from its dynamic features that need runtime handlers, can be an interesting MoC for NM4SMP performance evaluation.

BIBLIOGRAPHY

[1] Sergi Abadal, Albert Cabellos-Aparicio, Eduard Alarcon, and Josep Torrellas, « WiSync: An Architecture for Fast Synchronization through On-Chip Wireless Communication », *in*: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, Association for Computing Machinery, 2016, 3–17, ISBN: 9781450340915, DOI: 10.1145/2872362.2872396, URL: https://doi.org/10.1145/2872362.2872396.

[2] Jose L. Abell´n, Juan Fern´ndez, and Manuel E. Acacio, « GLocks: Efficient Support for Highly-Contended Locks in Many-Core CMPs », *in*: *2011 IEEE International Parallel Distributed Processing Symposium*, 2011, pp. 893–905, DOI: 10.1109/IPDPS.2011.87.

[3] Jose L. Abellán, Juan Fernández, and Manuel E. Acacio, « A G-Line-Based Network for Fast and Efficient Barrier Synchronization in Many-Core CMPs », *in*: *2010 39th International Conference on Parallel Processing*, 2010, pp. 267–276, DOI: 10.1109/ICPP.2010.34.

[4] Almutaz Adileh, Cecilia González-Álvarez, Juan Miguel De Haro Ruiz, and Lieven Eeckhout, « Racing to Hardware-Validated Simulation », *in*: *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019, pp. 58–67.

[5] A. Akram and L. Sawalha, « A Survey of Computer Architecture Simulation Techniques and Tools », *in*: *IEEE Access* 7 (2019), pp. 78120–78145.

[6] A. Akram and L. Sawalha, « ×86 computer architecture simulators: A comparative study », *in*: *2016 IEEE 34th International Conference on Computer Design (ICCD)*, 2016, pp. 638–645.

[7] Marco A. Z. Alves, Henrique C. Freitas, and Philippe O. A. Navaux, « Investigation of Shared L2 Cache on Many-Core Processors », *in*: *International Conference on Architecture of Computing Systems 2009*, 2009, pp. 1–10.

[8] Ibrahim A. Amory, Ahmed H. Ahmed, and Zahraa Hasan, « MESI protocol for multicore processors based on FPGA », *in*: *Periodicals of Engineering and Natural Sciences (PEN)* 9 (2021), pp. 80–89.

[9] James H. Anderson and Bojan Grošelj, « Pseudo read-modify-write operations: Bounded wait-free implementations », *in*: *Distributed Algorithms*, ed. by Sam Toueg, Paul G. Spirakis, and Lefteris Kirousis, Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 52–70.

[10] Jelena Antić, Georgios Chatzopoulos, Rachid Guerraoui, and Vasileios Trigonakis, « Locking Made Easy », *in*: *Proceedings of the 17th International Middleware Conference*, Middleware '16, Association for Computing Machinery, 2016, ISBN: 9781450343008, DOI: 10.1145/2988336.2988357, URL: https://doi.org/10.1145/2988336.2988357.

[11] *Architectural Exploration with gem5*, 2017, URL: http://gem5.org/ASPLOS2017_tutorial.

[12] Arvind, Kim P. Gostelow, and Wil Plouffe, « Indeterminacy, Monitors, and Dataflow », *in*: *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, SOSP '77, New York, NY, USA: Association for Computing Machinery, 1977, 159–169, ISBN: 9781450378673, DOI: 10.1145/800214.806559, URL: https://doi.org/10.1145/800214.806559.

[13] Mochamad Asri, Ardavan Pedram, Lizy K. John, and Andreas Gerstlauer, « Simulator calibration for accelerator-rich architecture studies », *in*: *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, 2016, pp. 88–95.

[14] T. Austin, E. Larson, and D. Ernst, « SimpleScalar: an infrastructure for computer system modeling », *in*: *Computer* 35.*2* (2002), pp. 59–67.

[15] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, « The NAS parallel benchmarks summary and preliminary results », *in*: *Supercomputing '91:Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, 1991, pp. 158–165, DOI: 10.1145/125826.125925.

[16] R. Banakar, S. Steinke, Bo-Sik Lee, M. Balakrishnan, and P. Marwedel, « Scratchpad memory: a design alternative for cache on-chip memory in embedded systems », *in*: *CODES 2002 (IEEE Cat. No.02TH8627)*, May 2002, DOI: 10.1145/774789.774805.

[17] Basilio B. Fraguela and Diego Andrade, « A software cache autotuning strategy for dataflow computing with UPC++ DepSpawn », *in*: *Comp. and Math. Methods* 1.*1* (Feb. 2021), pp. 1–14, DOI: 10.1002/cmm4.1148.

[18] Fabrice Bellard, « QEMU, a Fast and Portable Dynamic Translator », *in*: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, USENIX Association, 2005, p. 41.

[19] Luca Benini and Giovanni Micheli, « Networks on Chips: A new SoC paradigm », *in*: *Computer* 35 (Feb. 2002), pp. 70–78, DOI: 10.1109/2.976921.

[20] Shuvra S. Bhattacharyya, Ed F. Deprettere, and Bart D. Theelen, « Dynamic Dataflow Graphs », *in*: *Handbook of Signal Processing Systems*, ed. by Shuvra S. Bhattacharyya, Ed F. Deprettere, Rainer Leupers, and Jarmo Takala, New York, NY: Springer New York, 2013, pp. 905–944, ISBN: 978-1-4614-6859-2, DOI: 10.1007/978-1-4614-6859-2_28, URL: https://doi.org/10.1007/978-1-4614-6859-2_28.

[21] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li, « The PARSEC Benchmark Suite: Characterization and Architectural Implications », *in*: *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, ACM, 2008, pp. 72–81.

[22] Nathan Binkert et al., « The Gem5 Simulator », *in*: *SIGARCH Comput. Archit. News* 39.*2* (Aug. 2011), 1–7.

[23] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt, « The M5 Simulator: Modeling Networked Systems », *in*: *IEEE Micro* 26.*4* (July 2006), 52–60.

[24] Geoffrey Blake, Ronald G. Dreslinski, and Trevor Mudge, « A survey of multicore processors », *in*: *IEEE Signal Processing Magazine* 26.*6* (2009), pp. 26–37, DOI: 10.1109/MSP.2009.934110.

[25] Mark Bohr, « A 30 Year Retrospective on Dennard's MOSFET Scaling Paper », *in*: *IEEE Solid-State Circuits Society Newsletter* 12.*1* (2007), pp. 11–13.

[26] Adnan Bouakaz, Pascal Fradet, and Alain Girault, « A Survey of Parametric Dataflow Models of Computation », *in*: *ACM Transactions on Design Automation of Electronic Systems* (Jan. 2017), URL: https://hal.inria.fr/hal-01417126.

[27] Daniel P. Bovet and Marco Cesati, « Understanding the Linux kernel », *in*: *Understanding the Linux kernel*, 3rd, O'Reilly, 2006, chap. 10, p. 295.

[28] Wilfried Brauer and Wolfgang Reisig, « Carl adam Petri and "Petri nets" », *in*: *Fundamental concepts in computer science* 3.5 (2009), pp. 129–139.

[29] Joseph T. Buck, « Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model », PhD thesis, EECS Department, University of California, Berkeley, 1993.

[30] Anastasiia Butko, Rafael Garibotti, Luciano Ost, and Gilles Sassatelli, « Accuracy evaluation of GEM5 simulator system », *in*: *7th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, 2012, pp. 1–7, DOI: 10.1109/ReCoSoC.2012.6322869.

[31] T. E. Carlson, W. Heirman, and L. Eeckhout, « Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation », *in*: *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011, pp. 1–12, DOI: 10.1145/2063384.2063454.

[32] Trevor E. Carlson, Wim Heirman, Osman Allam, Stefanos Kaxiras, and Lieven Eeckhout, « The Load Slice Core microarchitecture », *in*: *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015, pp. 272–284, DOI: 10.1145/2749469.2750407.

[33] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, and Ibrahim Hurand Lieven Eeckhout, « An Evaluation of High-Level Mechanistic Core Models », *in*: *ACM Transactions on TACO* (2014).

[34] R. Cataldo, R. Fernandes, K. J. M. Martin, J. Sepulveda, A. Susin, C. Marcon, and J. Diguet, « Subutai: Distributed Synchronization Primitives in NoC Interfaces for Legacy Parallel-Applications », *in*: *Design Automation Conference*, ACM/ESDA/IEEE, 2018, pp. 1–6, DOI: 10.1109/DAC.2018.8465806.

[35] Donald D. Chamberlin, « The "Single-Assignment" Approach to Parallel Processing », *in*: *Proceedings of the November 16-18, 1971, Fall Joint Computer Conference*, AFIPS '71 (Fall), New York, NY, USA: Association for Computing Machinery, 1972, 263–269, ISBN: 9781450379106, DOI: 10.1145/1479064.1479114, URL: https://doi.org/10.1145/1479064.1479114.

[36] TravisS. Craig, *Building FIFO and Priority-Queuing Spin Locks from Atomic Swap*, tech. rep., University of Washington, 1993.

[37] Jack B. Dennis, « Data Flow Computer Architecture », *in*: *Encyclopedia of Parallel Computing*, ed. by David Padua, Boston, MA: Springer US, 2011, ISBN: 978-0-387-09766-4, DOI: 10.1007/978-0-387-09766-4_512, URL: https://doi.org/10.1007/978-0-387-09766-4_512.

[38] K. Desnos, « Memory Study and Dataflow Representations for Rapid Prototyping of Signal Processing Applications on MPSoCs », Theses, INSA de Rennes, Sept. 2014.

[39] Karol Desnos, Maxime Pelcat, Jean-François Nezan, and Slaheddine Aridhi, « On Memory Reuse Between Inputs and Outputs of Dataflow Actors », *in*: *ACM Trans. Embed. Comput. Syst.* 15.*2* (Feb. 2016), ISSN: 1539-9087, DOI: 10.1145/2871744, URL: https://doi.org/10.1145/2871744.

[40] Karol Desnos, Maxime Pelcat, Jean-François Nezan, and Slaheddine Aridhi, « Memory Analysis and Optimized Allocation of Dataflow Applications on Shared-Memory MPSoCs », *in*: *Journal of Signal Processing Systems* (July 2015), ISSN: 1939-8018, DOI: 10.1007/s11265-014-0952-6, URL: https://doi.org/10.1007/s11265-014-0952-6 (visited on 07/21/2020).

[41] L. Domagala, D. van Amstel, and F. Rastello, « Generalized Cache Tiling for Dataflow Programs », *in*: *SIGPLAN/SIGBED*, LCTES 2016, ACM, 2016, pp. 52–61, ISBN: 978-1-4503-4316-9, DOI: 10.1145/2907950.2907960, (visited on 08/29/2019).

[42] Michel Dubois, Murali Annavaram, and Per Stenstrm, *Parallel Computer Organization and Design*, USA: Cambridge University Press, 2012, ISBN: 0521886759.

[43] Lieven Eeckhout, « Computer Architecture Performance Evaluation Methods », *in*: *Synthesis Lectures on Computer Architecture* 5.*1* (2010), pp. 1–145.

[44] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger, « Dark silicon and the end of multicore scaling », *in*: *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, 2011, pp. 365–376.

[45] G. Estrin, B. Bussell, R. Turn, and J. Bibb, « Parallel Processing in a Restructurable Computer System », *in*: *IEEE Transactions on Electronic Computers* EC-12.*6* (1963), pp. 747–755, DOI: 10.1109/PGEC.1963.263558.

[46] Cesare Ferri, Amber Viescas, Tali Moreshet, R. Iris Bahar, and Maurice Herlihy, « Energy Efficient Synchronization Techniques for Embedded Architectures », *in*: *Proceedings of the 18th ACM Great Lakes Symposium on VLSI*, GLSVLSI '08, Association for Computing Machinery, 2008, 435–440, ISBN: 9781595939999, DOI: 10.1145/1366110.1366213, URL: https://doi.org/10.1145/1366110.1366213.

[47] M. France-Pillois, J. Martin, and F. Rousseau, « Implementation and Evaluation of a Hardware Decentralized Synchronization Lock for MPSoCs », *in*: *International Parallel and Distributed Processing Symposium*, IEEE, 2020, pp. 1112–1121, DOI: 10.1109/IPDPS47924.2020.00117.

[48] V. García, J. Gomez-Luna, T. Grass, A. Rico, E. Ayguade, and A. Pena, « Evaluating the effect of last-level cache sharing on integrated GPU-CPU systems with heterogeneous applications », *in*: *IEEE International Symposium on Workload Characterization (IISWC)*, IEEE, Sept. 2016, pp. 1–10, DOI: 10.1109/IISWC.2016.7581277.

[49] D. Genbrugge, S. Eyerman, and L. Eeckhout, « Interval simulation: Raising the level of abstraction in architectural simulation », *in*: *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, 2010, pp. 1–12.

[50] Alemeh Ghasemi, Rodrigo Cataldo, Jean-Philippe Diguet, and Kevin J. M. Martin, « On Cache Limits for Dataflow Applications and Related Efficient Memory Management Strategies », *in*: *Workshop on Design and Architectures for Signal and Image Processing (14th Edition)*, DASIP '21, Association for Computing Machinery, 68–76, ISBN: 9781450389013, DOI: 10.1145/3441110.3441573, URL: https://doi.org/10.1145/3441110.3441573.

[51] Alemeh Ghasemi, Marcelo Ruaro, Rodrigo Cataldo, Jean-Philippe Diguet, and Kevin J. M. Martin, « The Impact of Cache and Dynamic Memory Management in Static Dataflow Applications », *in*: *Journal of Signal Processing Systems* (Feb. 2022), ISSN: 1939-8115, DOI: 10.1007/s11265-021-01730-7, URL: https://doi.org/10.1007/s11265-021-01730-7.

[52] Christina Giannoula, Nandita Vijaykumar, Nikela Papadopoulou, Vasileios Karakostas, Ivan Fernandez, Juan Gómez-Luna, Lois Orosa, Nectarios Koziris, Georgios Goumas, and Onur Mutlu, « SynCron: Efficient Synchronization Support for Near-Data-Processing Architectures », *in*: *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 263–276, DOI: 10.1109/HPCA51647.2021.00031.

[53] James R. Goodman, « Retrospective: using cache memory to reduce processor-memory traffic », *in*: *ISCA '98*, 1998.

[54] Rachid Guerraoui, Hugo Guiroux, Renaud Lachaize, Vivien Quéma, and Vasileios Trigonakis, « Lock–Unlock: Is That All? A Pragmatic Analysis of Locking in Software Systems », *in*: *ACM Trans. Comput. Syst.* 36.*1* (Mar. 2019), ISSN: 0734-2071, DOI: 10.1145/3301501, URL: https://doi.org/10.1145/3301501.

[55] J. R. GURD, C. C. KIRKHAM, and I. WATSON, *ARTICLES THE MANCHESTER PROTOTYPE DATAFLOW COMPUTER*, USA: Communications of ACM, 1985.

[56] Anthony Gutierrez, Joseph Pusdesris, Ronald G. Dreslinski, Trevor Mudge, Chander Sudanthi, Christopher D. Emmons, Mitchell Hayenga, and Nigel Paver, « Sources of error in full-system simulation », *in*: *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014, pp. 13–22.

[57] Soonhoi Ha and Hyunok Oh, « Decidable Dataflow Models for Signal Processing: Synchronous Dataflow and Its Extensions », *in*: *Handbook of Signal Processing Systems*, ed. by Shuvra S. Bhattacharyya, Ed F. Deprettere, Rainer Leupers, and Jarmo Takala, New York, NY: Springer New York, 2013, pp. 1083–1109, ISBN: 978-1-4614-6859-2, DOI: 10.1007/978-1-4614-6859-2_33, URL: https://doi.org/10.1007/978-1-4614-6859-2_33.

[58] R. Hamzah and H. Ibrahim, « Literature Survey on Stereo Vision Disparity Map Algorithms », *in*: *Journal of Sensors* 16.*1* (Dec. 2015), pp. 1–23, DOI: 10.1155/2016/8742920.

[59] Muhammad Amber Hassaan, Martin Burtscher, and Keshav Pingali, « Ordered vs. Unordered: A Comparison of Parallelism and Work-Efficiency in Irregular Algorithms », *in*: *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, San Antonio, TX, USA: Association for Computing Machinery, 3–12, ISBN: 9781450301190, DOI: 10.1145/1941553.1941557, URL: https://doi.org/10.1145/1941553.1941557.

[60] Bijun He, William N. Scherer, and Michael L. Scott, « Preemption Adaptivity in Time-Published Queue-Based Spin Locks », *in*: *Proceedings of the 12th International Conference on High Performance Computing*, HiPC'05, Springer-Verlag, 2005, 7–18, ISBN: 3540309365, DOI: 10.1007/11602569_6, URL: https://doi.org/10.1007/11602569_6.

[61] M. Heinrich, V. Soundararajan, J. Hennessy, and A. Gupta, « A quantitative analysis of the performance and scalability of distributed shared memory cache coherence protocols », *in*: *IEEE Transactions on Computers* 48.*2* (1999), pp. 205–217, DOI: 10.1109/12.752662.

[62] J. Heulot, M. Pelcat, K. Desnos, J. Nezan, and S. Aridhi, « Spider: A Synchronous Parameterized and Interfaced Dataflow-based RTOS for multicore DSPS », *in*: *European Embedded Design in Education and Research Conference (EDERC)*, Milan, Italy: IEEE, 2014, pp. 167–171, DOI: 10.1109/EDERC.2014.6924381.

[63] C.J. Hughes, V.S. Pai, P. Ranganathan, and S.V. Adve, « Rsim: simulating shared-memory multiprocessors with ILP processors », *in*: *Computer* 35.*2* (2002), pp. 40–49.

[64] IEEE, « IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7 », *in*: *IEEE Std 1003.1-2017* 1.*1* (Jan. 2020), pp. 1–3951, DOI: 10.1109/IEEESTD.2018.8277153.

[65] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes*, English, version 325462-072US, Intel Corporation, 2020, 5052 pp., May, 2020.

[66] ITRS, International technology roadmap for semiconductors, the 2010 update, http://www.itrs.net (2011)., 2011.

[67] Bruce Jacob, Spencer Ng, and David Wang, *Memory Systems: Cache, DRAM, Disk*, Jan. 2008.

[68] Mark C. Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez, « A Scalable Architecture for Ordered Parallelism », *in*: *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, Association for Computing Machinery, 2015, 228–241, ISBN: 9781450340342, DOI: 10.1145/2830772.2830777, URL: https://doi.org/10.1145/2830772.2830777.

[69] Mark C. Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez, « Unlocking Ordered Parallelism with the Swarm Architecture », *in*: *IEEE Micro* 36.*3* (2016), pp. 105–117, DOI: 10.1109/MM.2016.12.

[70] Norman Jouppi, « Cache Write Policies and Performance. », *in*: vol. 21, May 1993, pp. 191–201, DOI: 10.1109/ISCA.1993.698560.

[71] Gilles Kahn, « The Semantics of a Simple Language for Parallel Programming », *in*: *Information Processing, Proceedings of the 6th IFIP Congress 1974, Stockholm, Sweden, August 5-10, 1974*, ed. by Jack L. Rosenfeld, North-Holland, 1974, pp. 471–475.

[72] John H. Kelm, Daniel R. Johnson, William Tuohy, Steven S. Lumetta, and Sanjay J. Patel, « Cohesion: A Hybrid Memory Model for Accelerators », *in*: *SIGARCH Comput. Archit. News* 38.*3* (June 2010), 429–440, ISSN: 0163-5964, DOI: 10.1145/1816038.1816019, URL: https://doi.org/10.1145/1816038.1816019.

[73] John H. Kelm, Matthew R. Johnson, Steven S. Lumettta, and Sanjay J. Patel, « WAYPOINT: Scaling Coherence to Thousand-Core Architectures », *in*: *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, Association for Computing Machinery, 2010, 99–110, ISBN: 9781450301787, DOI: 10.1145/1854273.1854291.

[74] T. Kim, Z. Sun, H. Chen, H. Wang, and S. X. . Tan, « Energy and Lifetime Optimizations for Dark Silicon Manycore Microprocessor Considering Both Hard and Soft Errors », *in*: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25.*9* (2017), pp. 2561–2574, DOI: 10.1109/TVLSI.2017.2707401.

[75] Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen, « Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors », *in*: *SIGARCH Comput. Archit. News* 35.*2* (2007), 162–173, ISSN: 0163-5964, DOI: 10.1145/1273440.1250683, URL: https://doi.org/10.1145/1273440.1250683.

[76] N. Kurd, P. Mosalikanti, M. Neidengard, J. Douglas, and R. Kumar, « Next Generation Intel¯ Core™ Micro-Architecture (Nehalem) Clocking », *in*: *IEEE Journal of Solid-State Circuits* 44.*4* (2009), pp. 1121–1129, DOI: 10.1109/JSSC.2009.2014023.

[77] Leslie Lamport, « How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs », *in*: *IEEE Transactions on Computers C-28* 9 (1979), pp. 690–691, URL: https://www.microsoft.com/en-us/research/publication/make-multiprocessor-computer-correctly-executes-multiprocess-programs/.

[78] Q.-T. Le, J. Stern, and S. Brenner, 2020.

[79] E.A. Lee and T.M. Parks, « Dataflow process networks », *in*: *Proceedings of the IEEE* 83.*5* (1995), pp. 773–801, DOI: 10.1109/5.381846.

[80] S. Lee, D. Tiwari, Y. Solihin, and J. Tuck, « HAQu: Hardware-accelerated queueing for fine-grained threading on a chip multiprocessor », *in*: *International Symposium on High Performance Computer Architecture*, IEEE, 2011, pp. 99–110, DOI: 10.1109/HPCA.2011.5749720.

[81] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St. Pierre, David S. Wells, Monica C. Wong, Shaw-Wen Yang, and Robert Zak, « The Network Architecture of the Connection Machine CM-5 (Extended Abstract) », *in*: *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, Association for Computing Machinery, 1992, 272–285, ISBN: 089791483X, DOI: 10.1145/140901.141883, URL: https://doi.org/10.1145/140901.141883.

[82] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, « McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures », *in*: *International Symposium on Microarchitecture (MICRO)*, New York, NY, USA: IEEE, 2009, pp. 469–480.

[83] Shang Li, Rommel Sánchez Verdejo, Petar Radojković, and Bruce Jacob, « Rethinking Cycle Accurate DRAM Simulation », *in*: *Proceedings of the International Symposium on Memory Systems*, MEMSYS '19, Association for Computing Machinery, 2019, 184–191.

[84] Ching-Kai Liang and Milos Prvulovic, « MiSAR: Minimalistic synchronization accelerator with resource overflow management », *in*: *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015, pp. 414–426, DOI: 10.1145/2749469.2750396.

[85] Mengxiao Liu, Weixing Ji, Zuo Wang, Jiaxin Li, and Xing Pu, « High Performance Memory Management for a Multi-core Architecture », *in*: *2009 Ninth IEEE International Conference on Computer and Information Technology*, vol. 1, 2009, pp. 63–68, DOI: 10.1109/CIT.2009.120.

[86] Pejman Lotfi-Kamran, Boris Grot, Michael Ferdman, Stavros Volos, Onur Kocberber, Javier Picorel, Almutaz Adileh, Djordje Jevdjic, Sachin Idgunji, Emre Ozer, and Babak Falsafi, « Scale-out Processors », *in*: *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, IEEE Computer Society, 500–511, ISBN: 9781450316422.

[87] D. G. Lowe, « Object recognition from local scale-invariant features », *in*: *IEEE International Conference on Computer Vision (ICCV)*, vol. 2, 1999, 1150–1157 vol.2, DOI: 10.1109/ICCV.1999.790410.

[88] A. Maghazeh, S. Chattopadhyay, P. Eles, and Z. Peng, « Cache-Aware Kernel Tiling: An Approach for System-Level Performance Optimization of GPU-Based Applications », *in*: *Design, Automation, and Test in Europe (DATE)*, DATE, IEEE, Mar. 2019, pp. 570–575, DOI: 10.23919/DATE.2019.8714861.

[89] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner, « Simics: A Full System Simulation Platform », *in*: *Computer* 35.*2* (Feb. 2002), 50–58, ISSN: 0018-9162.

[90] Neethu Bal Mallya and Cecilia González-Alvarez, « Flexible Timing Simulation of RISC-V Processors with Sniper », *in*: *In Proceedings of Second Workshop on Computer Architecture Research with RISC-V (CARRV. 2018)*, 2018.

[91] D. F. Martin and G. Estrin, « Path Length Computations on Graph Models of Computations », *in*: 18.*6* (1969), 530–536, ISSN: 0018-9340, DOI: 10.1109/T-C.1969.222705, URL: https://doi.org/10.1109/T-C.1969.222705.

[92] Kevin J. M. Martin, Mostafa Rizk, Martha Johanna Sepulveda, and Jean-Philippe Diguet, « Notifying Memories: A Case-Study on Data-Flow Applications with NoC Interfaces Implementation », *in*: *Design Automation Conference*, IEEE, 2016, p. 6, DOI: 10.1145/2897937.2898051.

[93] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood, « Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset », *in*: *SIGARCH Comput. Archit. News* 33.*4* (Nov. 2005), 92–99.

[94] Sally A. McKee and Robert W. Wisniewski, « Memory Wall », *in*: *Encyclopedia of Parallel Computing*, ed. by David Padua, Boston, MA: Springer US, 2011, pp. 1110–1116, ISBN: 978-0-387-09766-4, DOI: 10.1007/978-0-387-09766-4_234, URL: https://doi.org/10.1007/978-0-387-09766-4_234.

[95] John M. Mellor-Crummey and Michael L. Scott, « Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors », *in*: *ACM Trans. Comput. Syst.* 9.*1* (Feb. 1991), 21–65, ISSN: 0734-2071, DOI: 10.1145/103727.103729, URL: https://doi.org/10.1145/103727.103729.

[96] John M. Mellor-Crummey and Michael L. Scott, « Scalable Reader-Writer Synchronization for Shared-Memory Multiprocessors », *in*: *SIGPLAN Not.* 26.*7* (Apr. 1991), 106–113, ISSN: 0362-1340, DOI: 10.1145/109626.109637, URL: https://doi.org/10.1145/109626.109637.

[97] Jason E. Miller, Harshad Kasture, George Kurian, Charles Gruenwald, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal, « Graphite: A distributed parallel simulator for multicores », *in*: *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, 2010, pp. 1–12.

[98] M. Monchiero, G. Palermo, C. Silvano, and O. Villa, « Efficient Synchronization for Embedded On-Chip Multiprocessors », *in*: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 14.*10* (2006), pp. 1049–1062, DOI: 10.1109/TVLSI.2006.884147.

[99] Gordon E. Moore, « Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff. », *in*: *IEEE Solid-State Circuits Society Newsletter* 11.*3* (2006), pp. 33–35, DOI: 10.1109/N-SSC.2006.4785860.

[100] Fernando Moraes, Ney Calazans, Aline Mello, Leandro Möller, and Luciano Ost, « HERMES: an infrastructure for low area overhead packet-switching networks on chip », *in*: *Integration* 38.*1* (2004), pp. 69–93, ISSN: 0167-9260, DOI: https://doi.org/10.1016/j.vlsi.2004.03.003, URL: https://www.sciencedirect.com/science/article/pii/S0167926004000185.

[101] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, « Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0 », *in*: *International Symposium on Microarchitecture (MICRO)*, Chicago, IL, USA: IEEE, 2007, pp. 3–14, DOI: 10.1109/MICRO.2007.33.

[102] Stephen Neuendorffer and Edward Lee, « Hierarchical reconfiguration of dataflow models », *in*: *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE'04.* IEEE, 2004, pp. 179–188.

[103] Jung-Sub Oh, M. Prvulović, and A. Zajić, « TLSync: Support for multiple fast barriers using on-chip transmission lines », *in*: *2011 38th Annual International Symposium on Computer Architecture (ISCA)* (2011), pp. 105–115.

[104] Kunle Olukotun, Lance Hammond, and James Laudon, « Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency », *in*: *Synthesis Lectures on Computer Architecture* 2.*1* (2007), pp. 1–145.

[105] Peter S. Pacheco and Matthew Malensek, eds., *An Introduction to Parallel Programming*, Second Edition, Philadelphia: Morgan Kaufmann, 2022, p. iv, ISBN: 978-0-12-804605-0, DOI: https://doi.org/10.1016/B978-0-12-804605-0.00003-8.

[106] A. K. Panda, R. Palisetty, and K. C. Ray, « Area-Efficient Parallel-Prefix Binary Comparator », *in*: *2019 IEEE International Symposium on Smart Electronic Systems (iSES) (Formerly iNiS)*, 2019, pp. 12–16, DOI: 10.1109/iSES47678.2019.00016.

[107] Mark S. Papamarcos and Janak H. Patel, « A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories », *in*: *Proceedings of the 11th Annual International Symposium on Computer Architecture*, ISCA '84, New York, NY, USA: Association for Computing Machinery, 1984, 348–354, ISBN: 0818605383, DOI: 10.1145/800015.808204, URL: https://doi.org/10.1145/800015.808204.

[108] David A. Patterson and John L. Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*, 1st, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017, ISBN: 0128122757.

[109] M. Pelcat, K. Desnos, J. Heulot, C. Guy, J. Nezan, and S. Aridhi, « Preesm: A dataflow-based rapid prototyping framework for simplifying multicore DSP programming », *in*: *European Embedded Design in Education and Research Conference (EDERC)*, 2014, pp. 36–40, DOI: 10.1109/EDERC.2014.6924354.

[110] M. Pelcat, Karol Desnos, Julien Heulot, Clément Guy, J.F. Nezan, and Slaheddine Aridhi, « PREESM: A Dataflow-Based Rapid Prototyping Framework For Simplifying Multicore DSP Programming », *in*: Sept. 2014.

[111] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui, « The Tao of Parallelism in Algorithms », *in*: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, Association for Computing Machinery, 2011, 12–25, ISBN: 9781450306638, DOI: 10.1145/1993498.1993501, URL: https://doi.org/10.1145/1993498.1993501.

[112] PREESM, *PREESM Applications Repository*. Retrieved April 10, 2021, 2021, URL: https://github.com/preesm/preesm-apps.

[113] Claudius Ptolemaeus, ed., *System Design, Modeling, and Simulation using Ptolemy II*, Ptolemy.org, 2014.

[114] V. Rathore, V. Chaturvedi, A. Singh, T. Srikanthan, and M. Shafique, « Longevity Framework: Leveraging Online Integrated Aging-Aware Hierarchical Mapping and VF-Selection for Lifetime Reliability Optimization in Manycore Processors », *in*: *IEEE Transactions on Computers* (2020), pp. 1–1, DOI: 10.1109/TC.2020.3006571.

[115] V. Reddi, A. Settle, D. Connors, and R. Cohn, « PIN: a binary instrumentation tool for computer architecture research and education », *in*: *WCAE '04*, 2004.

[116] Stefan Reif, Benedict Herzog, Fabian Hügel, Timo Hönig, and Wolfgang Schröder-Preikschat, « Nearly Symmetric Multi-Core Processors », *in*: *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '20, Association for

Computing Machinery, 2020, 42–49, ISBN: 9781450380690, DOI: 10.1145/3409963.3410486, URL: https://doi.org/10.1145/3409963.3410486.

[117] Jose Renau, Basilio Fraguela, James Tuck, Wei Liu, Milos Prvulovic, Luis Ceze, Smruti Sarangi, Paul Sack, Karin Strauss, and Pablo Montesinos, *SESC simulator*, http://sesc.sourceforge.net.

[118] Daniel Sanchez and Christos Kozyrakis, « The ZCache: Decoupling Ways and Associativity », *in*: *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, USA: IEEE Computer Society, 2010, 187–198, ISBN: 9780769542997.

[119] Daniel Sanchez and Christos Kozyrakis, « ZSim: fast and accurate microarchitectural simulation of thousand-core systems », *in*: *ISCA' 13 Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013, pp. 475–486.

[120] Daniel Sanchez, Richard M. Yoo, and Christos Kozyrakis, « Flexible Architectural Support for Fine-Grain Scheduling », *in*: *SIGARCH Comput. Archit. News* 38.*1* (Mar. 2010), 311–322, ISSN: 0163-5964, DOI: 10.1145/1735970.1736055, URL: https://doi.org/10.1145/1735970.1736055.

[121] Ashley Saulsbury, Fong Pong, and Andreas Nowatzyk, « Missing the Memory Wall: The Case for Processor/Memory Integration », *in*: *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, ISCA '96, Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1996, 90–101, ISBN: 0897917863, DOI: 10.1145/232973.232984, URL: https://doi.org/10.1145/232973.232984.

[122] John E. Savage, *Models of Computation: Exploring the Power of Computing*, 1st, USA: Addison-Wesley Longman Publishing Co., Inc., 1997, ISBN: 0201895390.

[123] Patrick R. Schaumont, *A Practical Introduction to Hardware/Software Codesign*, 2nd, Springer Publishing Company, Incorporated, 2013.

[124] Hermann Schweizer, Maciej Besta, and Torsten Hoefler, « Evaluating the Cost of Atomic Operations on Modern Architectures », *in*: *2015 International Conference on Parallel Architecture and Compilation (PACT)*, 2015, pp. 445–456, DOI: 10.1109/PACT.2015.24.

[125] Michael Scott, « Shared-Memory Synchronization », *in*: *Synthesis Lectures on Computer Architecture* 8 (June 2013), pp. 1–221, DOI: 10.2200/S00499ED1V01Y201304CAC023.

[126] Joseph J. Sharkey, Dmitry Ponomarev, and Kanad Ghose, *M-Sim: A Flexible, Multithreaded Architectural Simulation Environment*, tech. rep., State University of New York at Binghamton, 2005.

[127] N. Slingerland and A. Smith, « Cache Performance for Multimedia Applications », *in*: *International Conference on Supercomputing (ICS)*, ICS '01, New York: ACM, 2001, pp. 204–217, ISBN: 978-1-58113-410-0, DOI: 10.1145/377792.377833, (visited on 06/24/2019).

[128] Yan Solihin, *Fundamentals of Parallel Multicore Architecture*, Nov. 2015, ISBN: 9781482211184, DOI: 10.1201/b20200.

[129] Daniel J. Sorin, Mark D. Hill, and David A. Wood, « A Primer on Memory Consistency and Cache Coherence », *in*: *Synthesis Lectures on Computer Architecture* 6.*3* (2011), pp. 1–212, DOI: 10.2200/S00346ED1V01Y201104CAC016, URL: https://doi.org/10.2200/S00346ED1V01Y201104CAC016.

[130] Sundararajan Sriram and Shuvra S Bhattacharyya, *Embedded multiprocessors: Scheduling and synchronization*, CRC press, 2018.

[131] « Front Matter », *in*: *High Performance Computing*, ed. by Thomas Sterling, Matthew Anderson, and Maciej Brodowicz, Boston: Morgan Kaufmann, 2018, p. iii, ISBN: 978-0-12-420158-3, DOI: https://doi.org/10.1016/B978-0-12-420158-3.01001-7, URL: https://www.sciencedirect.com/science/article/pii/B9780124201583010017.

[132] C. Stoif, M. Schoeberl, B. Liccardi, and J. Haase, « Hardware synchronization for embedded multi-core processors », *in*: *2011 IEEE International Symposium of Circuits and Systems*, IEEE, 2011, pp. 2557–2560, DOI: 10.1109/ISCAS.2011.5938126.

[133] Arthur Stoutchinin and Luca Benini, « StreamDrive: A Dynamic Dataflow Framework for Clustered Embedded Architectures », *in*: *J. Signal Process. Syst.* 91.*3–4* (Mar. 2019), 275–301, ISSN: 1939-8018, DOI: 10.1007/s11265-018-1351-1, URL: https://doi.org/10.1007/s11265-018-1351-1.

[134] S. Stuijk, M. Geilen, B. Theelen, and T. Basten, « Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications », *in*: *2011 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation* (2011), pp. 404–411.

[135] Lukasz Szustak, « Strategy for data-flow synchronizations in stencil parallel computations on multi-/manycore systems », *in*: *The Journal of Supercomputing* 74.*4* (), pp. 1534–1546, ISSN: 1573-0484, DOI: 10.1007/s11227-018-2239-3, URL: https://doi.org/10.1007/s11227-018-2239-3.

[136] J. Torrellas, H.S. Lam, and J.L. Hennessy, « False sharing and spatial locality in multiprocessor caches », *in*: *IEEE Transactions on Computers* 43.*6* (1994), pp. 651–663, DOI: 10.1109/12.286299.

[137] Andras Vajda, « Multi-core and Many-core Processor Architectures », *in*: *Programming Many-Core Chips.* (June 2011).

[138] Enrique Vallejo, Ramon Beivide, Adrian Cristal, Tim Harris, Fernando Vallejo, Osman Unsal, and Mateo Valero, « Architectural Support for Fair Reader-Writer Locking », *in*: *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, USA: IEEE Computer Society, 2010, 275–286, ISBN: 9780769542997, DOI: 10.1109/MICRO.2010.12, URL: https://doi.org/10.1109/MICRO.2010.12.

[139] *Vax macro and instruction set reference manual*, 2009, URL: https://www.ece.lsu.edu/ee4720/doc/vax.pdf.

[140] Rommel Sánchez Verdejo, Kazi Asifuzzaman, Milan Radulovic, Petar Radojkovic, Eduard Ayguadé, and Bruce Jacob, « Main memory latency simulation: the missing link », *in*: *MEMSYS*, 2018, pp. 107–116.

[141] Y. Wang, R. Wang, A. Herdrich, J. Tsai, and Y. Solihin, « CAF: Core to core Communication Acceleration Framework », *in*: *International Conference on Parallel Architecture and Compilation Techniques*, ACM, 2016, pp. 351–362, DOI: 10.1145/2967938.2967954.

[142] Y. Wang, R. Wang, A. Herdrich, J. Tsai, and Y. Solihin, « CAF: Core to core Communication Acceleration Framework », *in*: *International Conference on Parallel Architecture and Compilation Techniques*, ACM, 2016, pp. 351–362.

[143] Reinhold P. Weicker, « Dhrystone: A Synthetic Systems Programming Benchmark », *in*: *Commun. ACM* 27.*10* (1984), 1013–1030, ISSN: 0001-0782, DOI: 10.1145/358274.358283, URL: https://doi.org/10.1145/358274.358283.

[144] R.W. Wisniewski, L. Kontothanassis, and M.L. Scott, « Scalable spin locks for multiprogrammed systems », *in*: *Proceedings of 8th International Parallel Processing Symposium*, 1994, pp. 583–589, DOI: 10.1109/IPPS.1994.288245.

[145] Wm. A. Wulf and Sally A. McKee, « Hitting the Memory Wall: Implications of the Obvious », *in*: *SIGARCH Comput. Archit. News* 23.*1* (Mar. 1995), 20–24, ISSN: 0163-5964, DOI: 10.1145/216585.216588, URL: https://doi.org/10.1145/216585.216588.

[146] Joseph Yiu, *in*: *The Definitive Guide to Arm® Cortex®-M0 and Cortex-M0+ Processors*, Second, Oxford: Newnes, 2015, p. iv.

[147] Matt T. Yourst, « PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator », *in*: *2007 IEEE International Symposium on Performance Analysis of Systems Software*, 2007, pp. 23–34.

[148] Hervé Yviquel, « From dataflow-based video coding tools to dedicated embedded multi-core platforms », PhD thesis, University of Rennes 1, 2013.

[149] Weirong Zhu, Vugranam C Sreedhar, Ziang Hu, and Guang R. Gao, « Synchronization State Buffer: Supporting Efficient Fine-Grain Synchronization on Many-Core Architectures », *in*: *SIGARCH Comput. Archit. News* 35.*2* (June 2007), 35–45, ISSN: 0163-5964, DOI: 10.1145/1273440.1250668, URL: https://doi.org/10.1145/1273440.1250668.

DOCTORAT BRETAGNE LOIRE / MATHSTIC

Université Bretagne Sud

Titre : Mémoires notifiantes pour applications flux-de-données sur machines parallèles à mémoire partagée

Mot clés : Mémoire partagée, modèle flux-de-données, synchronisation

Résumé : Les machines parallèles à mémoire partagée (SMP) constituent une solution pratique pour mettre en œuvre des architectures multiprocesseurs puisqu'elles proposent une vue unifiée de la mémoire aux programmeurs ce qui facilite le développement des applications, au prix d'un mécanisme coûteux de cohérence de cache. Par ailleurs, les modèles de calcul flux-de-données offrent aux développeurs l'expressivité pour spécifier des applications complexes, en explicitant le parallélisme, permettant ainsi d'exploiter les ressources disponibles. Cependant, une implémentation d'une application flux-de-données sur SMP nécessite de nombreuses synchronisations qui impliquent la cohérence de cache et pénalisent les performances. Cette thèse s'intéresse à la compréhension des sources d'inefficacité dans l'exécution de ces applications et propose des techniques qui s'appuient sur la synchronisation exprimée dans le modèle pour en améliorer les performances. Tout d'abord, nous avons extrait les caractéristiques des applications selon plusieurs métriques, puis nous avons évalué deux techniques de gestion mémoire, Copy-on-Write et Non-Temporal Memory, pour soulager la pression sur la mémoire. Enfin, en contribution principale, nous proposons une unité matérielle spécialisée, proche de la mémoire, appelée NM4SMP (Notifying Memory for SMP) permettant d'accélérer les applications flux-de-données en y intégrant les règles de déclenchement des calculs. L'approche est validée sur des applications dites statiques et reconfigurables. Les résultats montrent une accélération de 1,23 et une économie d'énergie de 15% pour une plateforme basée sur des processeurs Intel et plusieurs applications réelles.

Title: Notifying Memories for Dataflow Applications on Shared-Memory Parallel Computer

Keywords: Memory, dataflow, synchronisation

Abstract: Symmetric Shared-memory multiprocessor (SMP) is the most widely used implementation of high-performance multi-core processors. It offers a uniform shared memory view that eases the development of parallel applications, but it requires cache-coherency management among the cores. Besides, dataflow Model of Computation helps the developers to specify complex applications with explicit parallelism to efficiently exploit the parallel resources of SMP. However, a dataflow application running on SMP requires high synchronization for data communication that stresses the cache memory and penalizes performance. Existing techniques for synchronization are not suited to dataflow as they are not aware of the model of computation. This thesis aims to deeply study dataflow applications' behavior on SMP and proposes novel techniques to speed them up. First, we evaluate dataflow application behavior based on several statistics. Second, we evaluate two memory techniques called Copy-on-Write and Non-Temporal Memory Transfer, to alleviate the memory footprint of dataflow applications on caches. Third, as our main contribution, we introduce an optimized hardware logic implemented near memory, Notifying Memory for SMP (NM4SMP) designed to speed up dataflow applications. Our solution improves synchronization of shared data by considering dataflow firing rules within the logic. A HW-SW co-design platform integrating NM4SMP is presented to support static and reconfigurable dataflow applications. Overall results show an average speedup of 1.23× and an average energy saving about 15%, assuming Intel SMP baseline system and real dataflow applications.