Distributed runtime system with global address space and software cache coherence for a data-flow task model

François Gindraud

▶ To cite this version:

François Gindraud. Distributed runtime system with global address space and software cache coherence for a data-flow task model. Data Structures and Algorithms [cs.DS]. Université Grenoble Alpes, 2018. English. NNT : 2018GREAM001 . tel-01891061

HAL Id: tel-01891061
https://theses.hal.science/tel-01891061v1

Submitted on 9 Oct 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Communauté UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

Spécialité : **Informatique**

Arrêté ministèriel : 25 mai 2016

Présentée par

François Gindraud

Thèse dirigée par **Fabrice Rastello**
et co-encadrée par **Albert Cohen**

préparée au sein du **Laboratoire d'Informatique de Grenoble**
dans l'**École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Système distribué à adressage global et cohérence logicielle pour l'exécution d'un modèle de tâche à flot de données

Thése soutenue publiquement le **11 janvier 2018**,
devant le jury composé de :

Madame Christine Morin
Directeur de Recherche, Inria, Président
Monsieur Fabrice Rastello
Directeur de Recherche, Inria, Directeur de thèse
Monsieur Albert Cohen
Directeur de Recherche, Inria, Co-Encadrant de thèse
Monsieur Lawrence Rauchwerger
Full Professor, Texas A&M University, Rapporteur
Monsieur Mikel Luján
Royal Society University Research Fellow, University of Manchester, Rapporteur
Monsieur Francesco Zappa Nardelli
Directeur de Recherche, Inria, Examinateur
Monsieur Frédéric Desprez
Directeur de Recherche, Inria, Examinateur
Monsieur Pierre Guironnet de Massas
Ingénieur, Kalray, Examinateur

Distributed runtime system with global address space and software cache coherence for a data-flow task model

---

Système distribué à adressage global et cohérence logicielle pour l'exécution d'un modèle de tâche à flot de données

---

François Gindraud

Abstract

Distributed systems are widely used in HPC (*High Performance Computing*). Owing to rising energy concerns, some chip manufacturers moved from multi-core CPUs to MPSoC (*Multi-Processor System on Chip*), which includes a distributed system on one chip.

However distributed systems – with distributed memories – are hard to program compared to more friendly shared memory systems. A family of solutions called DSM (*Distributed Shared Memory*) systems has been developed to simplify the programming of distributed systems. DSM systems include NUMA architectures, PGAS languages, and distributed task runtimes. The common strategy of these systems is to create a *global address space* of some kind, and automate network transfers on accesses to global objects. DSM systems usually differ in their interfaces, capabilities, semantics on global objects, implementation levels (hardware / software), ...

This thesis presents a new software DSM system called Givy. The motivation of Givy is to execute programs modeled as dynamic task graphs with data-flow dependencies on MPSoC architectures (MPPA). Contrary to many software DSM, the *global address space* of Givy is indexed by *real pointers*: raw C pointers are made global to the distributed system. Givy global objects are memory blocks returned by `malloc()`. Data is replicated across nodes, and all these copies are managed by a *software cache coherence protocol* called *Owner Writable Memory*. This protocol can relocate coherence metadata, and thus should help execute irregular applications efficiently. The programming model cuts the program into tasks which are annotated with memory accesses, and created dynamically. Memory annotations are used to drive coherence requests, and provide useful information for scheduling and load-balancing.

The first contribution of this thesis is the overall design of the Givy runtime. A second contribution is the formalization of the Owner Writable Memory coherence protocol. A third contribution is its translation in a model checker language (*Cubicle*), and correctness validation attempts. The last contribution is the detailed allocator subsystem implementation: the choice of real pointers for global references requires a tight integration between memory allocator and coherence protocol.

Résumé

Les architectures distribuées sont fréquemment utilisées pour le calcul haute performance (HPC). Afin de réduire la consommation énergétique, certains fabricants de processeurs sont passés d'architectures multi-cœurs en mémoire partagée aux MPSoC. Les MPSoC (*Multi-Processor System On Chip*) sont des architectures incluant un système distribué dans une puce.

La programmation des architectures distribuées est plus difficile que pour les systèmes à mémoire partagée, principalement à cause de la nature distribuée de la mémoire. Une famille d'outils nommée DSM (*Distributed Shared Memory*) a été développée pour simplifier la programmation des architectures distribuées. Cette famille inclut les architectures NUMA, les langages PGAS, et les supports d'exécution distribués pour graphes de tâches. La stratégie utilisée par les DSM est de créer un *espace d'adressage global* pour les objets du programme, et de faire automatiquement les transferts réseaux nécessaires lorsque ces objets sont utilisés. Les systèmes DSM sont très variés, que ce soit par l'interface fournie, les fonctionnalités, la sémantique autour des objets globalement adressables, le type de support (matériel ou logiciel), ...

Cette thèse présente un nouveau système DSM à support logiciel appelé Givy. Le but de Givy est d'exécuter sur des MPSoC (MPPA) des programmes sous la forme de graphes de tâches dynamiques, avec des dépendances de flot de données (*data-flow*). L'espace d'adressage global (GAS) de Givy est indexé par des *vrais pointeurs*, contrairement à de nombreux autres systèmes DSM à support logiciel : les pointeurs bruts du langage C sont valides sur tout le système distribué. Dans Givy, les objets globaux sont les blocs de mémoire fournis par `malloc()`. Ces blocs sont répliqués entre les nœuds du système distribué, et sont gérés par un *protocole de cohérence de cache logiciel* nommé *Owner Writable Memory*. Le protocole est capable de déplacer ses propres métadonnées, ce qui devrait permettre l'exécution efficace de programmes irréguliers. Le modèle de programmation impose de découper le programme en tâches créées dynamiquement et annotées par leurs accès mémoire. Ces annotations sont utilisées pour générer les requêtes au protocole de cohérence, ainsi que pour fournir des informations à l'ordonnanceur de tâche (spatial et temporel).

Le premier résultat de cette thèse est l'organisation globale de Givy. Une deuxième contribution est la formalisation du protocole Owner Writable Memory. Le troisième résultat est la traduction de cette formalisation dans le langage d'un *model checker* (*Cubicle*), et les essais de validation du protocole. Le dernier résultat est la réalisation et explication détaillée du sous-système d'allocation mémoire : le choix de pointeurs bruts en tant qu'index globaux nécessite une intégration forte entre l'allocateur mémoire et le protocole de cohérence de cache.

Contents

Abstract iii

Résumé iv

Contents v

1 Introduction 1
- 1.1 High Performance Computing . . . 1
  - 1.1.1 Hardware Performance . . . 1
  - 1.1.2 Shared Memory vs Distributed Memory . . . 5
- 1.2 Distributed Shared Memory . . . 7
  - 1.2.1 Classification . . . 8
  - 1.2.2 Families . . . 10
- 1.3 Givy . . . 13
- 1.4 Outline & Contributions . . . 15

2 Runtime 17
- 2.1 Virtual Memory . . . 17
  - 2.1.1 Common Uses . . . 17
  - 2.1.2 Implementation . . . 19
  - 2.1.3 Posix API . . . 19
  - 2.1.4 In Givy . . . 20
- 2.2 Givy . . . 20
  - 2.2.1 Task . . . 20
  - 2.2.2 Scheduling . . . 22
  - 2.2.3 Allocator . . . 26
  - 2.2.4 Coherence Protocol . . . 27
- 2.3 Implementation . . . 27
  - 2.3.1 Version 1 . . . 27
  - 2.3.2 Version 2 . . . 28
  - 2.3.3 Network . . . 29
- 2.4 Extensions . . . 30
  - 2.4.1 Region Split / Merge . . . 30
  - 2.4.2 Region Set . . . 31
  - 2.4.3 Region Mutex . . . 31
  - 2.4.4 Stack and Globals Support . . . 32
  - 2.4.5 Code Section Support . . . 32

2.5 State of the Art . . . 32

3 Coherence Protocol **37**
3.1 Formalization Flavors . . . 37
3.1.1 Operational . . . 37
3.1.2 Axiomatic . . . 38
3.2 Cache Coherence Protocols 101 . . . 40
3.2.1 VI . . . 41
3.2.2 MSI - Snooping . . . 42
3.2.3 MSI - Directory . . . 44
3.2.4 MSI - Distributed . . . 44
3.3 Givy Coherence Protocol . . . 44
3.3.1 Formal Model . . . 45
3.3.2 Discussion and Extensions . . . 51

4 Memory Allocator **53**
4.1 Memory Allocation . . . 53
4.1.1 API . . . 54
4.1.2 Performance Criteria And Trade-offs . . . 56
4.1.3 Classic Allocator Structures . . . 57
4.2 Global Address Space Management . . . 59
4.3 Allocators – State of the Art . . . 61
4.4 Allocator V1 . . . 62
4.4.1 Page Mapper . . . 63
4.4.2 Page Mapper Base . . . 64
4.4.3 Page Mapper Fixed . . . 64
4.4.4 Page Mapper Auto . . . 66
4.4.5 Local Allocator . . . 68
4.4.6 Proxy Allocator . . . 70
4.4.7 Measurements . . . 70
4.4.8 Proxy Allocator Performance . . . 74
4.4.9 Conclusion . . . 77
4.5 Allocator V2 . . . 77
4.5.1 Basic Allocator Structure . . . 77
4.5.2 GAS Memory Management . . . 79
4.5.3 GAS Coherence Metadata . . . 79
4.5.4 Conclusion . . . 80

5 Correctness **81**
5.1 Cubicle . . . 81
5.1.1 Description . . . 81
5.1.2 Modeling . . . 83
5.1.3 Conclusion . . . 89
5.2 Shared Memory Data Structures . . . 90
5.2.1 C11 / C++11 Memory Model . . . 91
5.2.2 Cuckoo Hash Table . . . 92

6 Conclusion **97**

List of Figures **101**

List of Listings **103**

Bibliography **105**

Chapter 1

Introduction

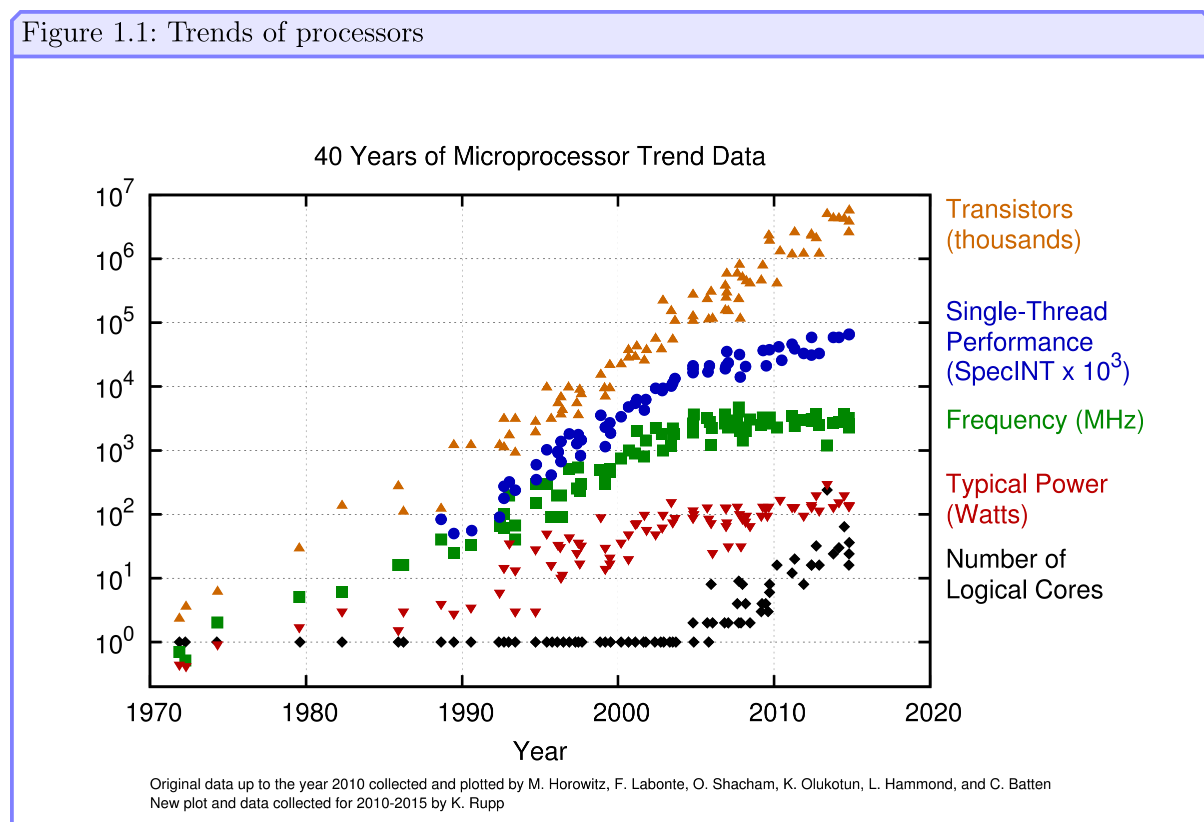
This chapter presents an overview of the challenges of high performance computing. It quickly describes the concept of Givy from a user point of view. It also places Givy into the current family of solutions to the HPC challenges.

1.1 High Performance Computing

1.1.1 Hardware Performance

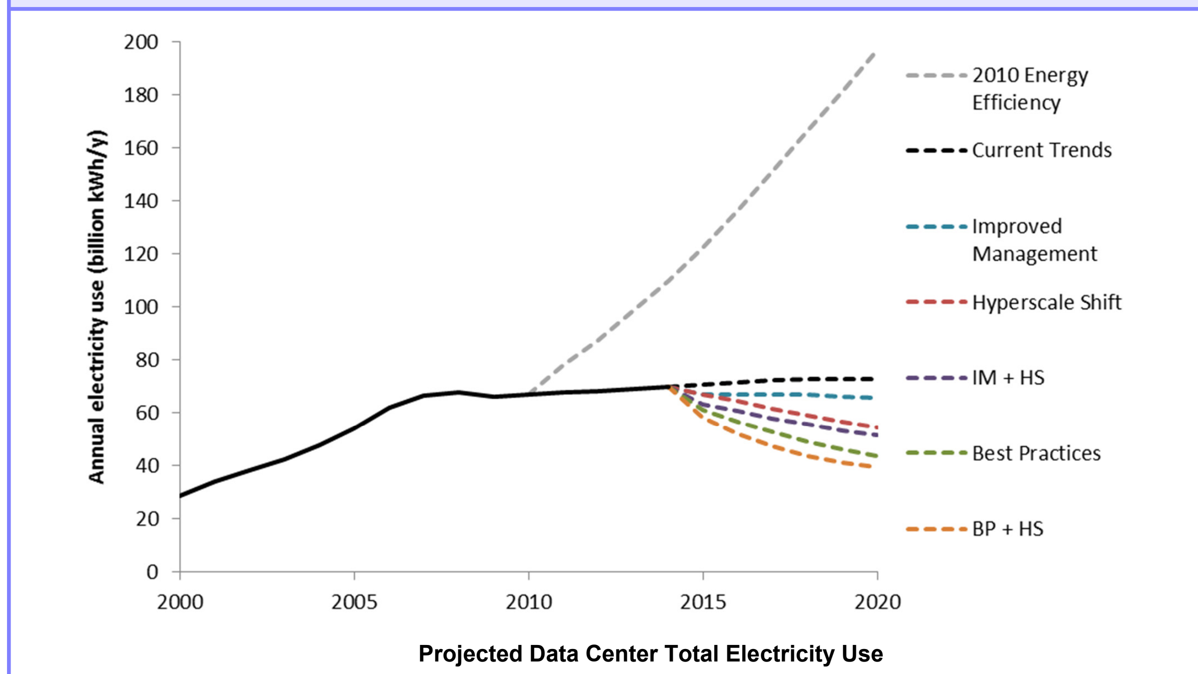
Processor frequencies have stagnated since around 2005. And *Moore's law* may start to fail after holding for years, as transistors starts to reach the size of a few atoms.

Figure 1.1: Trends of processors



During this period, processor vendors were still able to increase performance. But they couldn't increase the frequency. It would have added more and more constraints on chip design (keeping the clock signal coherent across the chip). It also would have increased energy consumption and dissipation. A low consumption is becoming increasingly important with the presence of smartphones and other battery powered embedded devices. And as computing is now using a significant percentage of the world energy expenditures, this is also important for the environment. Heat dissipation is also a problem, as data centers today spend a lot of energy in cooling systems.

Figure 1.2: Estimations of the energy consumption of US datacenters. Estimates include energy used for servers, storage, network equipment, and infrastructure in all U.S. data centers. The solid line represents historical estimates from 2000-2014 and the dashed lines represent five projection scenarios through 2020.



Instead, processor vendors used the increased transistor density to put more transistors on each chip. They increased single processor performance by adding more complexity to the processor internal structures. It includes: more *cache* storage (to reduce time spent waiting for data), deeper *instruction pipelines* (which increases instruction throughput but not latency), smarter *branch prediction* (to reduce branching impact on pipeline latency). And they also added parallelism, in the form of *vector instructions*, and by adding multiple cores on each chip.

The total available computing power has indeed increased. But being able to use it is now very difficult, as you must consider and play along with all the processor complex structures. In addition to that, memory speeds have been slow compared to processor improvements. So another big problem of today is to bring enough data to the processor. If the data to be processed does not arrive in time, the processor will stall and wait for it, wasting computing power. The *cache* memory is used to store a subset of the whole system memory with a low latency processor access. Thus keeping the processor fed with data requires a good cache management.

The advent of multi-core processors did not help with the memory and cache problem. Most of them use the *shared memory* model, in which they all access the same memory. This means that memory and cache throughput must be divided between the cores. In some cases increased parallelism can lead to worse performance due to memory and synchronization contention. In addition to that, memory and the caches must be kept *coherent* between the processor cores. Today's processors have up to 3 hierarchical levels of cache, some of them shared between some cores, and some private to cores. This means that keeping the performance *and* the memory correctness is very difficult, and a trade-off.

In addition to the evolution of single core processor to multi-core processors, other computing design have gained popularity. Most of them use the huge amount of available transistors to provide performance in their own way. They all present a different API from single core processors.

GPU GPUs (*Graphics Processing Unit*) started as accelerators for video game screen rendering. They have gained more flexibility over the years (see the name GPGPU, for *General Purpose GPU*).

Contrary to multi-core processors, they have high parallelism, and contain hundreds or thousand of small computing units. These units have a small instruction set. They are grouped in groups which will execute the same code on different data (like a huge vector instruction). Memory accesses are carefully categorized, between regular ones and irregular, and each will use a different caching system.

This hardware design is extremely efficient for executing the same operation on a large input set, like computing pixel color for each pixel of a screen. Today they are also able to execute more irregular computations with limited control flow, but lose some efficiency. A lot of current research is being done to relax the control flow constraints without hurting performance too much.

GPUs are still considered accelerators. So they are controlled from another processor that manages memory transfers and start computations. Current GPU are programmable with a small DSL (domain specific language), and require the use of a specific API for control. Some recent GPUs allow a limited *shared memory* model through *virtual memory mapping.*

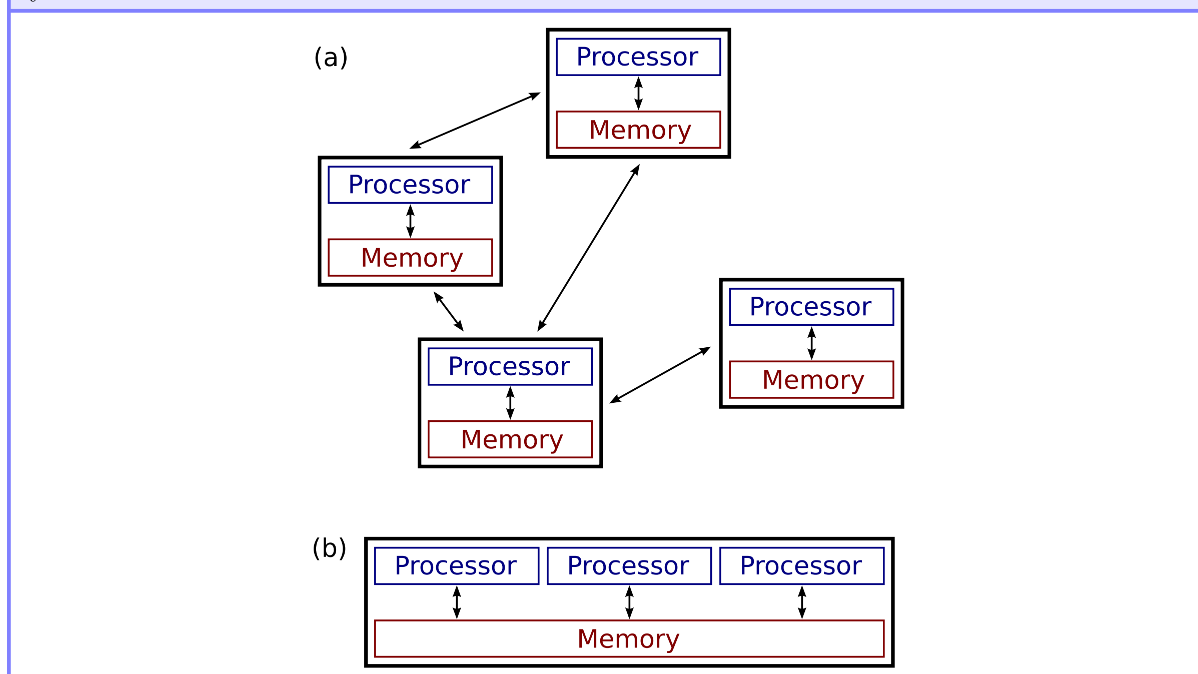
Computational Memory (C-RAM) C-RAM (*Computational Memory*, also called *near-memory computing* or processor-in-memory) is a recent alternative to classic processor-centric designs. Instead of improving the memory system around processing units (like in multi-core CPUs and GPUs), they add numerous small processing cores in the memory fabric. They share many similarities with GPU: similar to huge vector instructions, require external control with a specific API, huge number of small basic cores. They can be more efficient than GPUs (especially energy-wise), as there is no need to move the data in specific caches. However they add constraints due to memory geometry and data placement (memory banks, virtual memory, shared access with the controller CPU to memory).

FPGA FPGA (*Field Programmable Gate Array*) are programmable logic circuits. The huge amount of transistors is used to create a grid of programmable wires, wire con-

nections, and logic circuit units. A specific circuit can be designed and programmed on the chip. This is a cheaper and modifiable alternative to ASICs (*Application-Specific Integrated Circuit*), but provides less speed than them.

Again, they are often used as an accelerator, and require a specific API if linked a host machine for data transfer and control. Programming them requires designing a circuit using hardware description languages (VHDL) or higher level tools. As with hardware design, they are adapted to very regular code (fixed memory usage, static data structures). Some act as a programmable logic glue and integrate specific units in the grid, like memory banks, small processor cores, or specific computing units.

Figure 1.3: (a) Represents a Distributed memory system. (b) Represents a Shared memory system.



Distributed Systems Distributed systems are built by taking multiple *nodes*, and liking them by a network connection. Each node is a computer by itself, with an addressable memory, and computing elements (multi-core processor, sometimes coupled with a GPU): see Fig. 1.3.

These systems are usually built to increase the computing power when a single multi-core processor is not enough. Having independent distributed memories instead of a single shared memory makes hardware design easier and scales better. However they are very difficult to program, due to the presence of multiple addressable memories (one on each node). Data transfers between nodes must be performed manually (network transfers), and computation split between nodes (scheduling & synchronization problems).

Distributed systems include *grid computing clusters* and *super-computers*. Grid computing can be made from consumer grade hardware, while super computers use more specific hardware (especially for the network layer). Distributed systems are wide spread in HPC (High Performance Computing), because they scale well and because the API

of individual nodes are similar to that of well known multi-core processors (contrary to GPU, FPGA, or C-RAM).

There are many tools and frameworks trying to ease the programming of distributed systems. This thesis describes a new tool called Givy, in the family of the DSM (*Distributed Shared Memory*) systems (see Section 1.2).

MPSoC Distributed systems have started to be used in an embedded context due to energy concerns. Maintaining the coherence of shared memory costs energy, which is saved if memories are separate like in a distributed system. It is also easier to use energy-saving strategies like DVFS (*Dynamic Voltage Frequency Scaling*) or partial shutdown of cores in a distributed system (the nodes are not synchronous).

Thus companies have started to develop MPSoC (*Multi-Processor System on Chip*) [70], which includes a complete distributed system in one chip. These MPSoC form a middle ground between GPUs (high parallelism, but basic cores) and multi-core CPUs (low parallelism, complex cores). They have roughly the same properties as distributed systems, but are smaller: very small memories, slower cores.

Early MPSoC like RAW [67] had proper distributed memories without hardware coherence. More recent chips such as Tile-GX [10] or the Intel Xeon Phi [5] use a network internally, but local memories are hidden and act as caches (non addressable). This thesis was initially focused on a recent chip develop by the Kalray company [6], named the MPPA [33, 32]. The MPPA has 16 homogeneous nodes each with a 2MB addressable memory and 16 homogeneous cores (+1 for management). It also has virtual memory support (see Section 2.1) on each node. It is thus a proper distributed system on a chip, with the additional constraint of very low node memory.

1.1.2 Shared Memory vs Distributed Memory

In this section we will discuss further the differences between shared and distributed memory. A summary of the key differences is listed in Fig. 1.4.

Figure 1.4: Summary of properties of distributed and shared memories.

Property	**Shared**	**Distributed**
Addressing	global (`void*`)	local to nodes
Data transfer	automatic	manual
Programmer use	"easy"	hard
Hardware complexity	hard	easy
Energy consumption	high	low
Scaling	hard	easy

Addressing A shared memory is easier to program, as all cores *share the same address space*. It means that every program object has a single reference that is valid for all cores (a *pointer* in the C language). Objects can be linked together through these references, forming complex structures like graphs. And these structures can be understood as is by all cores without the need for any translation.

In a distributed system, each node has its exclusive *address space* (its node-local memory). A program object only exists one node. It can be copied to other nodes, or interact with other nodes by reacting to network messages. But it cannot be accessed directly like a pointer could be dereferenced in shared memory. Classic algorithmic structures like graphs or trees are very difficult to create without global references.

Data Transfer Transferring data between nodes in a distributed memory system is manual and cumbersome (example in Listing 1.1). Data must be transferred through the network, which require actions by both sender and receiver. The sender must know the destination, and package the data from its address space to a format that the network can handle (*serialization*). The receiver must also know how to handle the incoming message: how to put its data in its own address space (*deserialization*), and where to put it. This can be done by a blocking call (`recv()` in the example), or other techniques like non blocking calls or RDMA (see Section 2.3.3). Although a lot of frameworks like MPI [37] simplify this, a data transfer is a complex operation that requires careful coordination between nodes.

Listing 1.1: Data transfer in distributed memory

```
{ // Node 1
  int fd = connection_to_node2 ();
  int data [42];
  produce_data(data);
  send (fd, data, 42 * sizeof (int));
}
{ // Node 2
  int fd = connection_to_node1 ();
  int data [42];
  receive (fd, data, 42 * sizeof (int));
  use_data (data);
}
```

In shared memory, transfers are mostly automatic, as in Listing 1.2. A piece of data doesn't need to be translated to any intermediate format, and has a global reference that both threads can use. As soon as data is written by `produce_data()`, it will automatically be "transferred" to other threads. To make these transfers predictable, *synchronization primitives* must be used (like `pthread_barrier_wait()` in the example). `pthread_barrier_wait()` blocks until both threads have reach their calls, and ensures that all data has been transferred before returning. Other synchronization primitives exist: *mutexes*, *atomics* (see Section 5.2.1), *memory barriers*.

Cache Coherence Protocol If the shared memory system has a *true* shared memory, all writes to memory would be instantaneously visible (transferred) to any other threads. However, most shared memory processors use multiple levels of caches, with the first level being usually private to each core. In this case, how are new memory writes automatically "transferred" to other cores (and their caches) ?

Listing 1.2: Data transfer in shared memory (naive example)

```
pthread_barrier_t barrier;
int data[42];
{ // Thread 1
  produce_data (data);
  pthread_barrier_wait (&barrier);
}
{ // Thread 2
  pthread_barrier_wait (&barrier);
  use_data (data);
}
```

Caches are kept synchronized through a *cache coherence protocol*. Caches can communicate on the chip, and notify other caches of new writes, to let them get the new values instead of using their stale stored values. The protocol describes how they communicate. It creates a semantic of when data is transferred, which is called *memory model* (see Section 5.2.1). *Memory barriers* – special processor instructions – are used to *flush* caches and force propagation of values.

This automatic notification and transfer system does not come with zero cost. It is much more complicated to design than distributed memories (protocol implementation, communication lines between caches). The communications between caches cost a lot of energy, and are difficult to scale to a high number of cores. Even with all these challenges, hardware manufacturers continue to provide shared memory, as it is so convenient for programmers [47].

1.2 Distributed Shared Memory

Definition The idea of DSM (*Distributed Shared Memory*) systems is to provide a programming model close to shared memory on top of a distributed memory hardware. DSM systems come with a wide variety of implementation choices, making them sometimes difficult to differentiate from a big hardware cache system. In this thesis, we define DSM systems as systems which:

- provide a shared-memory-like abstraction to programmers: global addressing, and global accesses;
- are built on top of a distributed memory system: addressable local memories, user visible network.

This definition excludes classic hardware caches from DSM systems, as they have non-addressable private memories (caches), and hidden networks (communication lines between caches). This also excludes chips like the Intel Xeon Phi [5], which has an internal hidden network that is used to create a distributed L2 abstraction. Most DSM systems are implemented with software, but not all (see NUMA systems, Section 1.2.2).

DSM vs MPI/OpenMP Like shared memory systems, a DSM will somewhat automate data transfers between distributed memories. Compared to a manually perfectly optimized message passing program implementation, a DSM will likely make unnecessary transfers and be far from optimal. The rules guiding the data transfers (memory model and coherence protocol) will over-approximate the data transfers needed (as under-approximating leads to a correctness problem).

But DSM are used to let programmers be more productive: using the DSM is easier, with a fast development cycle. Finding a perfect message passing solution can take years for expert programmers. And this perfect solution will likely include classic concepts like caching data. While it needs to be reinvented with message passing, DSM often already use it internally.

Another advantage of DSM are portability. Assuming DSM developers ported it to multiple platforms, programs using the DSM will be easier to port to these platforms. Message passing solutions are more dependent on machine features.

DSM can also hide the underlying architecture. On classic computing clusters made from multi-core nodes, traditional approaches use MPI [37] for inter-node communication, and OpenMP [31, 56] for multi-core management. While this is usable, it lets programmer handle difficult question like which cores can use the message passing system, under which synchronization scheme, or how to avoid dead-locks between the two... DSM usually have to solve these questions internally, moving the difficulty from application programmers to the DSM programmers.

DSM vs Hardware Coherence A software DSM will usually react slower than a completely hardware coherence system. However it is at a higher level of abstraction, and can use a wider range of information to optimize data transfers. For example, software DSMs can know the transferred object size, know if an object is node-local or globally shared, etc. Hardware implementation often use a "one size fits all" strategy, with fixed granularity, and will try to infer intent from the access pattern, which is harder. DSM can also be adapted faster and at lower cost than hardware.

1.2.1 Classification

In this section we will try to classify the variety of DSM systems.

Support Level The *support level* is the entity (or group of entities) that implement the DSM mechanisms:

- Hardware: data transfers are managed completely by hardware; mostly NUMA systems (Section 1.2.2);
- Compiler: data transfers are hidden by language features, compiled to data transfers calls (or API calls);
- Library: data transfers are managed by a library (function calls).

Hardware support is the least portable, as you need the specific hardware to make the program work. However it is also the most transparent, as it will likely behave like a

big multi-core CPU (NUMA systems). It is close to a hardware cache system, and share its flaws. It is often combined with a software API for high level operations (often an optimized implementation of MPI [7]).

Compiler strategies are often combined with a library one. The compiler will generate data transfer by adding calls to the library API. User programs must be programmed in the specific language.

Library support is the most portable and can fit in a existing language. However, users will have to add library calls manually.

Data References To support a shared-memory-like abstraction, the DSM system must provide a kind of *global reference*. These references are associated to a *global address space* (GAS). References can come in different types:

- Real pointer: a plain C pointer (`void*`), which can be used as is (mostly found in NUMA);
- Fake pointer type: a small type that requires translation before accessing the data (often a combination of node id and a local C pointer);
- Index / key in a table: global data looks like a table or array, and is accessed through a global key;
- Object: global data is defined as objects, which have methods calls for interaction.

Data Granularity The DSM system must handle piece of data of different sizes. Data granularity describes how the DSM will handle these different sizes:

- Fixed granularity: a size is chosen (cache line size, or page size), and the DSM operates on chunks of this size;
- Constant after creation: the DSM operates on the piece of data as a unit, but it cannot be resized;
- Dynamic: the DSM operates on the piece of data as a unit, and it can grow or shrink.

Fixed granularity is simple, but cannot adapt to the size of global objects. Objects smaller than the granularity size will generate bigger data transfers than necessary, and may suffer for *false sharing* if multiple objects are in the same chunk. Objects bigger than the granularity size may be transferred piece by piece instead of using one long transfer that can be optimized.

Data Management and Access DSM can manage access to global data in two ways:

- Replication and caching: the DSM maintains multiple copies of the data object on multiple nodes;
- Remote access: only one node maintains a copy, and other nodes request temporary copies of the data from it every time.

The difference between the two access models is not always clear cut, as in most case the remote node must copy the data in its own virtual address space before accessing it. NUMA machines tend to rely on their fast hardware and use remote access: the data might be reloaded many times. Some low level GAS frameworks (like ARMCI [52]) tend to get a local copy of remote data, modify it, and push it back: this is still remote access as the temporary buffer is not kept coherent with remote data. Replication DSM will internally manage the local copy, and ensure it is kept up to date before every access.

Execution Model Our last important criteria for categorization is what execution model is supported by the DSM system:

- Threaded: the program is a set of concurrent threads;
- Task graph: the parallel program is described as a set of tasks, linked by value flow and dependencies;

A message passing program is usually tied to the threaded model, with blocking network calls, and heavy ordering and coordination. Old DSM approaches [15] automated the network transfers but kept the threaded model. NUMA architectures also work under this model as they try to look like a multi-core machine.

Most of the recent DSM work instead uses the task graph method. Programs are described as a set of tasks (inputs, outputs, and processing code). These tasks inputs and outputs are linked, and the links represents value flow. The task graph is then executed according to these links (*dependencies*). Tasks not transitively dependent may be executed concurrently.

The task model requires a heavier run-time support than a classic threaded one. However, it is able to hide network transfers (or memory access) latency by executing other ready tasks while it moves data. These models are attractive, but are less programmer friendly as they are less structured than the threaded model.

Threaded models cannot easily move computation units between nodes, so they can lose efficiency if the computation is imbalanced between nodes. Task graph models can move the already split computation units (tasks) easily between nodes. However they have to manage the *scheduling*, i.e. choosing where and when tasks are executed. Although some systems choose the scheduling at compile time [68], most will choose it at run-time. A run-time scheduler must carefully balance the quality of the schedule and the overhead cost of the scheduling algorithm.

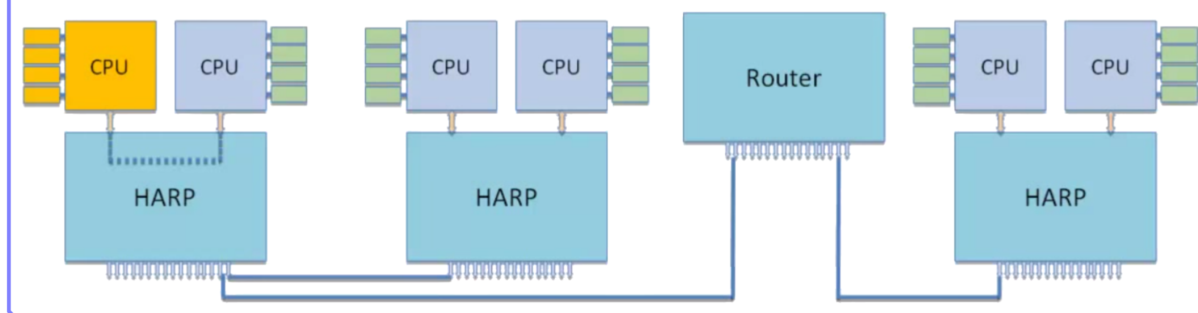
For both models, the program must split his work into a number of units that is adapted to the target machine number of processing units. Too few big units will underutilize the machine, and too many small units will loose efficiency in too much synchronization. Thus even if the task graph model looks “independent” from the machine, part of the graph generation must be able to scale according to the machine parallelism too.

1.2.2 Families

A DSM could be created with any set of the previously shown properties. However, over time DSM systems have formed a few big families.

NUMA Systems NUMA (*Non-Uniform Memory Access*) systems (Fig. 1.5) have special hardware to track memory accesses and generate network transfers if the data is remote. This hardware may be located in the CPU, in the cache hierarchy (cache coherent NUMA, or ccNUMA), in its off-chip memory system (a.k.a. chipset), and/or in the network interfaces.

Figure 1.5: Structure of an SGI UV-2000 NUMA machine. *Extracted from https://www.youtube.com/watch?v=Lk8nRO0dJlA*



NUMA systems allow to use unmodified shared memory programs. However, if programs can be ported without effort, the performance cannot. If no special care is given to the memory layout of the program, false sharing may occur. Due to low or absence of data replication, data is best created on the node that will access it the most. Reductions should use temporary per-node accumulators values on separate DSM chunks. Thus performance still requires heavy tuning [38]. To alleviate some of the problems due to the rigid nature of hardware, most NUMA hardware vendor also implement optimized message passing libraries [7].

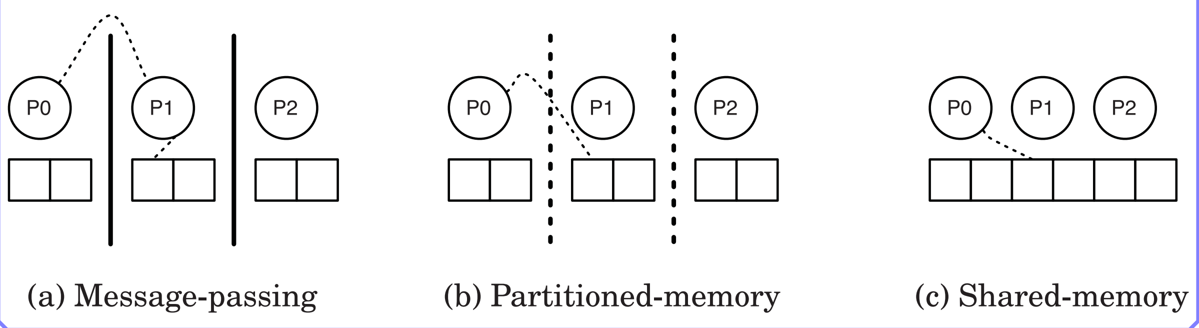
The *Software NUMA* systems are a sub family of NUMA. These DSM systems have a NUMA like behavior, but they detect memory accesses through the virtual memory system (Section 2.1). They do not require specific hardware and can be used in any system with virtual memory. However they suffer from a high fixed granularity, at the page level (usually 4KB). Examples include Treadmarks [15], and a DSM system for the MPPA at Kalray.

PGAS Languages The PGAS (*Partitioned Global Address Space*) language DSM family contains over a dozen different DSM systems (see [34] for a detailed overview).

PGAS sits between the shared and distributed models. It creates a global address space that can be accessed by all nodes, but it also keeps a *node-local* memory (see Fig. 1.6). Memory accesses in local and global memories use different constructs, so the distinction is visible to the programmer. This distinction is supposed to let the programmer be aware of the cost of accessing global memories, and avoid careless use like what could happen in a NUMA machine. Most PGAS languages try to use RDMA (Section 2.3.3) for remote accesses, as it does not require actions by the remote node (hence the dotted line avoids P1 in the figure).

Listing 1.3 shows some excerpts from UPC [26], a PGAS programming language. By our classification, UPC is a Compiler + Library, Threaded, Constant after creation granularity, Fake Pointer approach. UPC is a super-set of the C language, and it adds

Figure 1.6: PGAS compared to shared and distributed memories. P0 wants to access the same chunk of memory in every scenario, the dotted line represents the path taken by the transfer.



(a) Message-passing (b) Partitioned-memory (c) Shared-memory

a `shared` storage-specifier keyword to place variables in the global address space. It is supported by a run-time library, and the UPC compiler will automatically generate library calls when `shared` variables are accessed. Pointers into the GAS are fake pointers hidden by the compiler.

Listing 1.3: Example of UPC constructs

```
int a[16]; // local variable
shared int b[16]; // GAS variable
shared [4] int c[16]; // with explicit layout
b[3] += 42; // generates communication code
upc_memcpy(b, c, 16 * sizeof (int)); // GAS version of memcpy
shared int * gas_ptr; // fake pointer (node, virtual addr, phase)
```

A downside to these approaches is that the notion of a local memory is usually tied to a threaded model. This distinction between local and global memory also requires different libraries. In UPC, part of the C standard library has been duplicated into a GAS version (example: `upc_memcpy`).

Other PGAS languages include X10 [28], CoArray Fortran [55], Global Arrays [53], or Chapel [27]. They are usually based on libraries like ARMCI [52] or GasNet [24] which handle the system setup for the global address space.

Distributed Task Run-times Another family consists of the task graph DSM run-times. Examples include Stanford Legion [20], Intel Concurrent Collections (CnC) [25], StarSs [60] or Grappa [50].

Task DSM systems are based on a Task execution model, and most take the form of a library. StarSs uses OpenMP-like annotations with pragmas, and Legion can be either through its library or with a small language with a compiler pass. The library based ones require to fit the data to their structures with specific types, and often use index / key addressing, or fake pointers. Functions calls must be made to access piece of data in these tables, and they are used to detect accesses and make the necessary memory transfers.

The different run-times will be examined in more detail in Section 2.5.

Distributed Objects For completeness, we must mention the family of *distributed object* systems. In these DSM systems, *objects* are created and referenced in a global naming space. Interaction with these objects can be performed through methods, which will transfer arguments and retrieve results as needed. This family includes the CORBA standard [2], and the more recent trend of web APIs.

1.3 Givy

In this thesis, we wanted to develop a new DSM solution, which is called Givy. The initial motivation behind Givy was to build a DSM run-time to efficiently execute irregular parallel applications on MPSoC architectures. We also wanted to require as little program modification as possible. The Kalray MPPA was our main primary target, but we also support x86 clusters.

According to the previous classification, Givy is a Library support, Real pointer, Constant after creation granularity, Replication and Task based DSM system.

Having real C pointers (`T*`) as global references is rare in software DSMs and more often found in NUMA strategies. In Givy, we provide an unmodified data interface for the user. They do not have to rewrite pointer based data structures to fit the DSM like in other software DSMs, as pointers are valid in the global address space. However, currently only heap data is in the GAS (data and pointers in the space managed by `malloc()`/ `free()` and derivatives); stack and static data are not supported. This restriction is necessary as we need to have knowledge and control of the layout of program data. To control heap pointer values, we also require that the target architecture has virtual memory.

Using Givy still requires modifying the program, as we need to cut it into tasks. There is no way to bypass this modification if we want to balance irregular computations dynamically. Cutting the program into tasks gives a benefit: tasks inputs (function arguments) can be used to detect memory accesses.

Execution Model Givy programs are built following a dynamic data-flow execution model. A data-flow application is a dynamically built and unfolded graph of run-to-completion tasks with explicit data dependencies. Tasks are created at run-time by predecessors, and their dependencies are explicitly set by older tasks. When all dependencies are fulfilled and its data is available locally, a task may execute (non preemptively) and then its local resources be deallocated. Tasks are one-shot (they are destroyed after use) and cannot be interrupted (thus all accessed data must be visible in the input argument list).

Tasks arguments may be simple values (passed by value) or mutable memory *regions*. These regions are referenced through raw pointers. Each task accessing a region needs to specify the access mode, read-only (`const T*`) or read-write (`T*`). We require that the task graph corresponding to the target data-flow application is *data-race free* with respect to these memory regions. Extensions will allow some racy code patterns to be supported.

To execute such applications on distributed systems, the run-time system must be extended to deal with distributed memory. All memory references (*regions*) are placed in a global address space, so they can be accessed from any node. The run-time can perform

load balancing by migrating tasks either between cores or nodes. Each time a task will be executed, Givy will transfer the required data (input memory regions, and task frame with arguments) to its execution node. The global memory regions are replicated across nodes, and managed by a *software cache coherence protocol*.

Sparse Matrix Multiplication Let us consider the blocked-sparse matrix multiplication $C = A \times B$ as a motivating (dynamic) application. Sparse linear algebra is interesting because it is irregular, and use pointer indirections to sub blocks. An excerpt of C code describes a possible Givy task implementation in Listing 1.4, and Fig. 1.7 illustrates the shape of task dependencies.

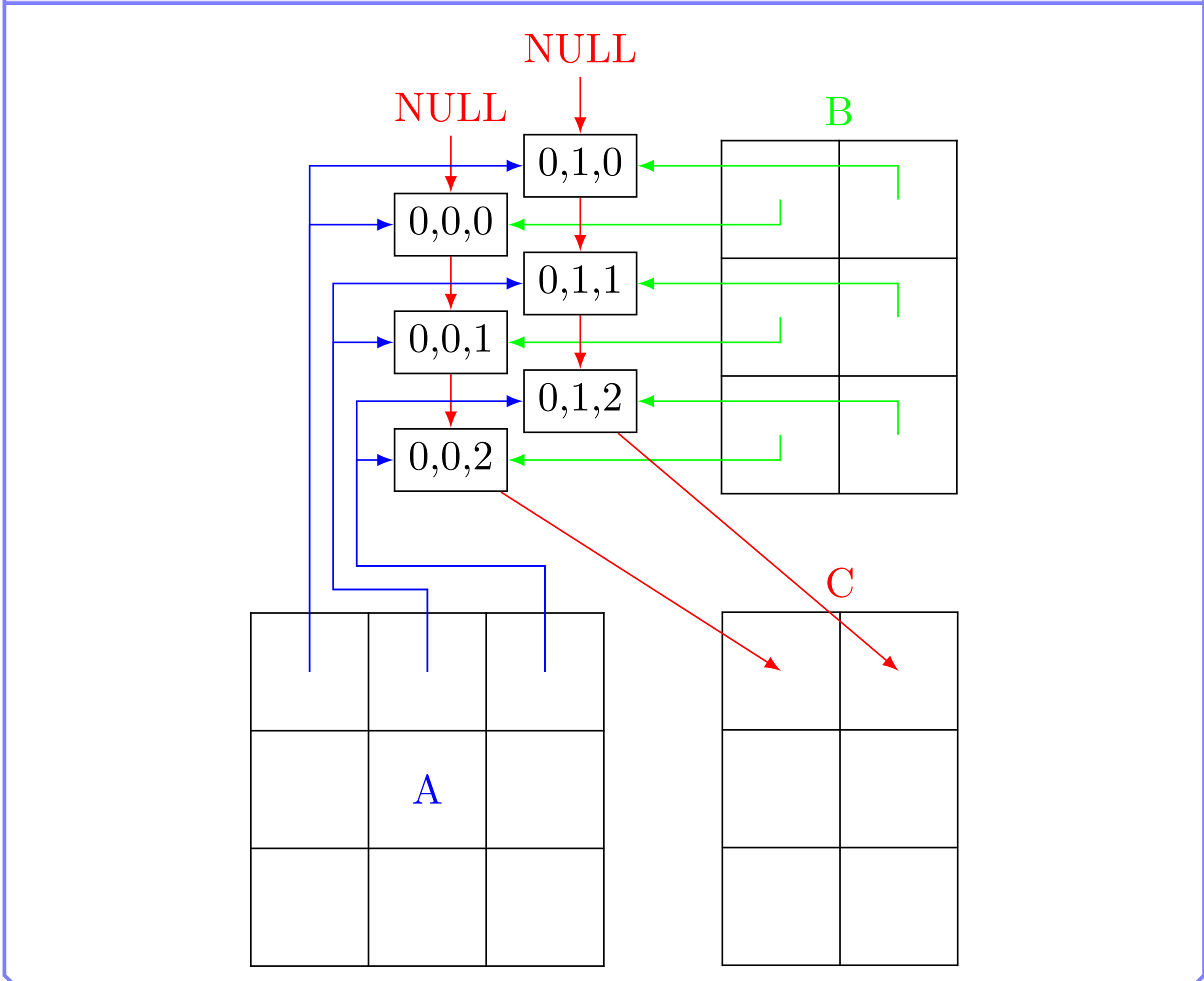
Listing 1.4: Givy example: blocked sparse matrix multiplication $C = A \star B$

```
typedef double* Matrix; // NULL <=> full of zeroes
Matrix A[M][K] = {...}, B[K][N] = {...}, C[M][N] = {NULL};
// ...
// will perform C[i][j] += A[i][k] * B[k][j]
void mm_chunk(TASK next_chunk, const double *a,
              const double *b, double *c)
{
  if (a != NULL && b != NULL) {
    if (c == NULL) { c = calloc(MATRIX_CHUNK_SIZE); }
    c += a * b; // dense matmul like cblas_gemm
  }
  // forward c value to the next mm_chunk task
  TASK_SET_ARG(next_chunk, c, 3 /* arg_pos=3 */);
}
// ...
// spawn all tasks (creates a chain of task for each i,j)
for (int i = 0; i < M; i++) {
  for (int j = 0; j < N; j++) {
    TASK next = gather_result_instance;
    for (int k = K - 1; k >= 0; k--)
      next = TASK_SPAWN(mm_chunk, next, A[i][k], B[k][j], UNDEF);
  }
}
```

Function `mm_chunk` is instantiated as $M \times N \times K$ tasks that perform the individual block dense matrix multiplication. Arguments `a` and `b` of each task are fixed at task creation as they do not change. Argument `c` represents `C[i][j]` which starts at NULL but might be allocated during execution to store a non-zero sub-matrix. So all tasks using `C[i][j]` are ordered as a chain and forward the current `c` to the next task in chain (pointed to by `next_chunk`, fixed at spawn). Forwarding the argument with `TASK_SET_ARG` also unlocks the next task which enters the scheduler as it now has all its arguments.

When the task is scheduled to run on a specific node, we check that each region argument is available in the requested access mode by reading the coherence meta-data. If this is not the case, we send the coherence requests on the network, and put the task

Figure 1.7: Givy task graph example: blocked sparse matrix multiplication $C = A \star B$



on wait and execute other tasks. When all coherence requests have been completed, the task can run.

1.4 Outline & Contributions

This thesis presents four contributions to the design of a new DSM. The first contribution is the design of Givy, a run-time that fulfills the following criteria: Library support, Real pointer, Constant after creation granularity, Replication and Task based. The second contribution is a formal design of a coherence protocol and its integration with the run-time. The third contribution is the translation of the formal protocol into a model checker language. The fourth contribution is a partial implementation of Givy. It mainly consists of a memory allocator that: allows GAS-level `malloc()`; provides support for replicated regions. The implementation is not complete enough to test performance of the whole Givy runtime system. Only partial performance evaluation of the memory allocator has been done.

The rest of the manuscript falls into the following four chapters before concluding. Chapter 2 describes the run-time design. Chapter 3 describes the cache coherence pro-

tocol formal model. Chapter 4 describes the implementation of the memory allocator. Chapter 5 describes the validation attempts of the coherence protocol with a model checker and discusses the correctness of the low-level concurrent implementation.

Chapter 2

Runtime

This chapter describes the overall design of Givy. Givy relies heavily on *virtual memory* and a *network layer*. Section 2.1 and Section 2.3.3 discuss these two concepts before diving into the details of Givy in Section 2.2. Section 2.5 compares Givy to other task run-times.

Region In the following chapters, a *region* is a dynamically allocated buffer that is managed by the coherence protocol in Givy.

Core A processor core, a single threaded computing unit. Each core has a worker thread that can execute tasks.

Node A node contains a memory (and its own address space), a processor with one or more cores that share the node memory, a network interface. The coherence mechanisms manage regions between node's memory spaces.

Target architecture Givy targets a system made of multiple nodes that can communicate through a network.

2.1 Virtual Memory

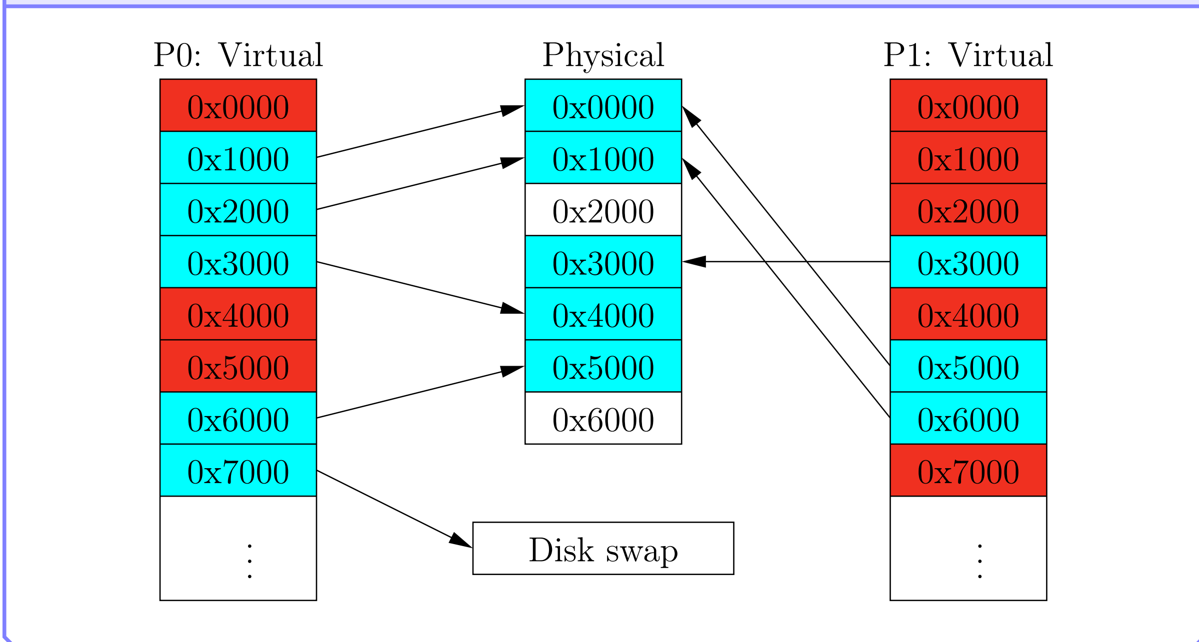
This section describes what is virtual memory, along with some hardware / OS details that are relevant to Givy.

Most general purpose CPU cores today have *virtual memory* support. When using virtual memory, two address spaces exists: *virtual addresses* used in program space, and *physical addresses* in computer memory. These address spaces may have different shapes, so an *address translation* step is needed before any memory access, from the virtual address to the physical one. The translation table is not required to be complete: if the virtual address is not found, the access will fail.

2.1.1 Common Uses

Most operating systems have exclusive control over the virtual memory system. They use it for many purposes: abstracting physical memory, security, disk swapping, error detection, memory sharing between processes, etc. Fig. 2.1 illustrates some of these, in a small system with two processes P0 and P1.

Figure 2.1: Example of a virtual memory layout for two processes P0 and P1. Virtual memory: light blue pages are mapped (usable), red ones unmapped (unusable). Physical memory: light blue pages are in use by either P0 or P1, white ones are not. Arrows represent translations.



Error detection Memory accesses without any translation entry generate errors. So there is a high chance that a bogus memory access will fall outside the set of current translations. Operating systems will catch the resulting error and stop the process (segmentation fault). Most operating systems always leave no translations for address `0x0` (`NULL`) to ensure that an access to `NULL` fails.

Physical memory abstraction The physical memory is not always contiguous, and may contain holes. Virtual memory can hide these holes by using a contiguous virtual space translation scheme.

Disk swapping Virtual memory can be used to simulate a huge memory, even if the physical memory is small. If the program working set is too big to fit the physical memory, some of it is stored away in slower storage devices. In consumer computers, this is called *disk swapping*, as the hard drive is used to store data from the RAM.

With virtual memory, stored data is marked as *available* even if it is not in the physical memory (see address `0x7000` of P0 in Fig. 2.1). Stored data has a special tag in the translation table. When accessed, the operating system can retrieve the data from storage, place it in physical memory, and complete the memory access as if nothing happened.

Security Virtual memory provides security by placing each process in a sandbox (with respect to memory accesses). Operating systems will create one translation table (and thus one virtual address space) per process. As long as this table does not contain translations to other processes or OS memory, the current process is completely isolated

in memory. For example, in Fig. 2.1, address `0x3000` refers to different physical memory in P0 and P1.

Sharing memory between processes Multiple translation tables can point to the same physical memory addresses, like physical `0x0000` and `0x1000` in Fig. 2.1. A common use case is read-only shared data, like binary code of libraries used by both processes. Another use case is when processes explicitly request to share some memory.

2.1.2 Implementation

Virtual memory support is a team work between the hardware and the operating system. The operating system stores the set of virtual-to-physical address translations in a structure called *page table* [40]. Each memory access must be intercepted, and its address translated using the table before accessing the physical memory.

In order to prevent too much overhead, the translation is performed by a hardware unit called *Memory Management Unit* (MMU). The MMU only translates the upper bits of the memory address: translation is done at a coarse granularity. A *page* is a set of addresses with the same upper bits relevant for the architecture MMU. On x86 the page size is fixed at 4KB. On the MPPA it is a configurable power of two starting at 4KB. This coarse granularity prevents the page table data structure from taking too much memory space (less entries for the same amount of mapped memory).

To speed up the translation even more, most MMUs use a translation cache called *Translation Lookaside Buffer* (TLB). This cache stores a few entries from the page table, and must be kept in sync with it by the operating system (TLB flush on table modifications). A TLB cache miss is called a *page fault*. It will usually trigger an interrupt, and let operating system code determine the missing translation. A bogus access (like `NULL`) will be detected by the operating system, and transformed into a *segmentation fault*. Modern TLB can use multiple cache levels, or more often provide multiple page sizes to better support high memory usage (4KB / 2MB / 1GB pages on x86). Carefully using the higher page sizes can prevent slowdowns due to TLB saturation, but require careful memory placement (aligning virtual memory to the higher page size).

> MMU also implement memory access protection. Each page has access mode flags (read / write / executable), that are compared to the accessing instruction and will trigger a segmentation fault if a mismatch happen.

Most operating systems nowadays lazily allocate physical pages to virtual mappings. Creating virtual memory in the OS will just create some page table entries. A physical page will be associated to a virtual page only when it is first accessed (first page fault).

2.1.3 Posix API

The first (and now deprecated) way to manipulate virtual memory was the `brk()` system call. It was used to extend or shrink a linear contiguous program heap. It lacks flexibility, and will easily lead to fragmentation, and thus most modern memory allocators avoid using it (except the libc one).

The preferred way to create / destroy virtual memory chunks is by using the `mmap()` / `munmap()` system calls. `mmap()` primary use is to create a virtual memory mapping that is backed by a unix *file descriptor*: the memory will be filled with the resource data as needed by the operating system. Memory allocators create *anonymous mappings* (without any file descriptor) using the `MAP_ANONYMOUS` flag: the memory is just a chunk of contiguous virtual memory pages freely usable. As we focus on memory allocation, in the remainder of this thesis `mmap()` is always used in the anonymous mode.

Some other system calls like `madvise()` and `mprotect()` can manipulate the created mapping flags and provide optimization hints to the operating system. However, there is no way to access the page table directly.

2.1.4 In Givy

Due to the use of raw C pointer as GAS addresses, C pointers must be carefully chosen to prevent collisions. Thus memory objects must be created at specific places in memory. Givy uses the virtual memory for this purpose, by implementing a `malloc()` that returns correctly placed pointers.

2.2 Givy

The Givy run-time contains 3 main components: a memory allocator, a cache coherence subsystem, and a task scheduler. These components interact with each other, with the *operating system* (system calls) and with *worker threads* (Givy calls). A summary of interactions can be found in Fig. 2.2.

As the run-time has not been completed, part of this section may describe my thoughts about how to implement the feature rather than an actual implementation. The Section 2.3 describes what has been implemented precisely.

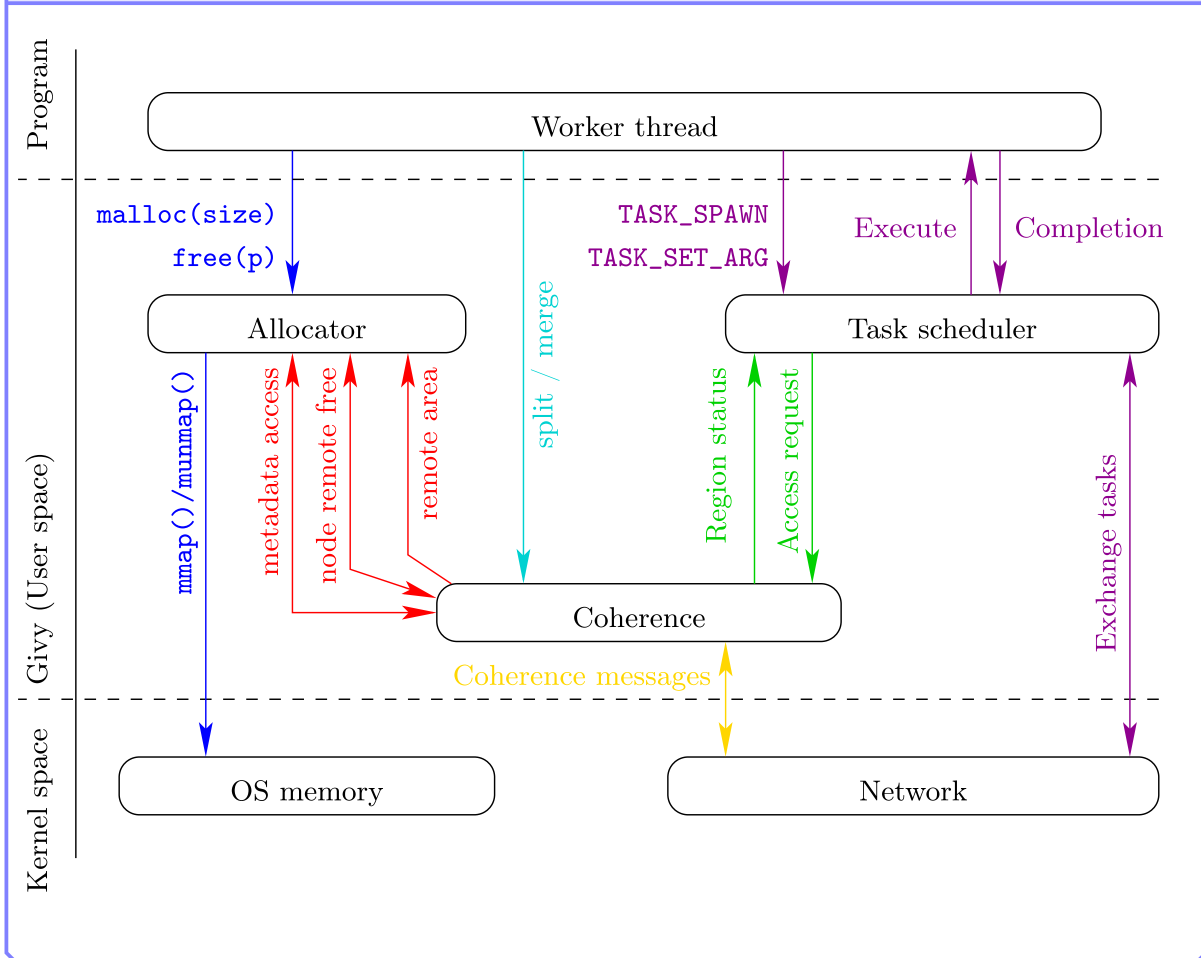
2.2.1 Task

We start with the most basic component of the runtime: a single task. In Givy, a task is a *closure* built around a C function. In other words, it is a C function, with a set of argument values that can be used to call the function, and also some metadata for the runtime.

Implementation wise, a task is represented by a *task frame*. A *task frame* is an instance of a C structure filled with task data. An example is given in Fig. 2.3, for a fictional function `f`. The most basic part of the task frame is a pointer to function `f`, to be able to call it when executing the task. In Givy task frames are dynamically allocated for each task, and are thus *memory regions*. The `TASK` pseudo-type is a pointer to the corresponding task frame, which is a unique identifier for the task due to our real pointer GAS choice.

The task frame contains storage space for each function argument. The primitive `TASK_SET_ARG` will locate the task frame and fill these argument values. Each argument is also annotated with a flag that determine its kind. A `VALUE` flag indicates a trivial type that should just be copied. A `READ_ONLY` or `READ_WRITE` flag indicates a *memory region*, i.e. a memory buffer that is managed by the coherence protocol. The access mode

Figure 2.2: Main components of Givy



determines which mechanism to trigger in the coherence protocol. Lastly, the `TASK` flag indicates that the value is a task handle (internally, this is a region with a special access mode).

The `SC` field is called *synchronization counter*. This field represents the number of dependencies that a task is still waiting for, before being able to run. It is initialized to some positive integer at task frame creation. It can be decremented, and when it reaches 0 the task is now eligible to be run. `TASK_SPAWN` will initialize the SC to the number of undefined arguments left in the task frame. `TASK_SET_ARG` will decrement the SC on each call, and thus make the task able to run when it reaches 0.

> Task frames do not store link to neighbors in the task graph. They only store SC as dependency information. The dependency information is contained in the task code, by calling `task_tdec` on the right tasks.

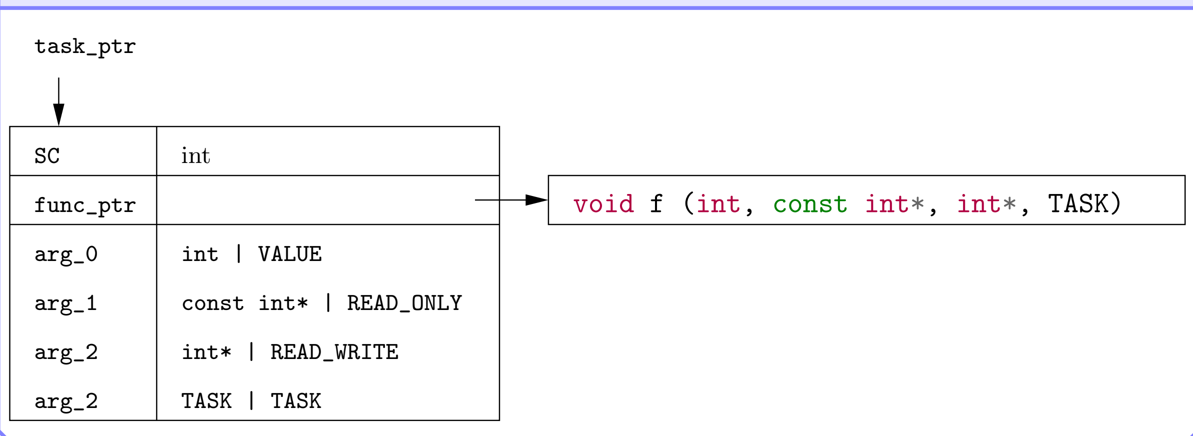


Figure 2.3: Structure of a task frame

The `TASK_SPAWN` / `TASK_SET_ARG` is a simplified API for the sake of presentation. Givy is designed more as a run-time target for higher level task computing systems (like OpenMP[31] or OpenStream [61]) than a user level run-time. Its actual API is lower level, and can trivially implement the simplified one :

`task* task_create(int sc, ArgInfo info[])`	Creates task frame
`void task_set_arg(task* t, ArgInfo arg, T value)`	Only sets argument
`void task_dec(task* t)`	Decrements SC

In particular, SC is decoupled from the number of argument, and can be used to represent additional dependencies that could be difficult to represent through arguments.

In terms of implementation complexity, `TASK_SPAWN` (and `task_create`) are simple because they just create a local region and sets some values. `TASK_SET_ARG`, implemented by `task_set_arg` and `task_tdec` is much more complex as it might need to access a task frame on remote nodes. They use a special mode of the cache coherence to do that efficiently. They are described in Section 3.3.2.

2.2.2 Scheduling

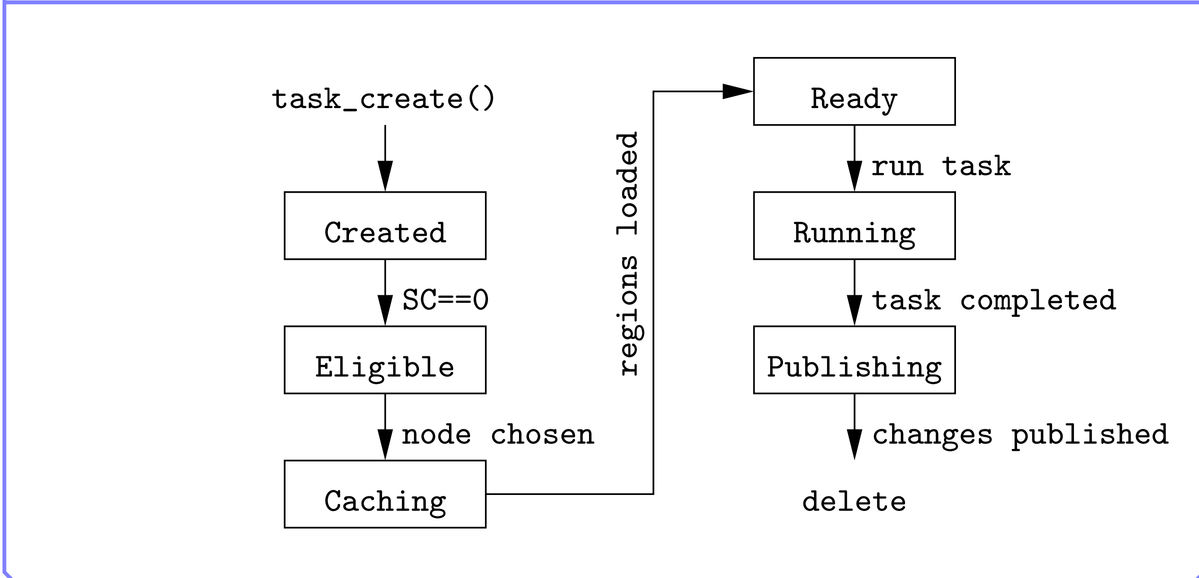
The next part of Fig. 2.2 to describe is the scheduler component. The scheduler is in charge of executing the tasks at the right time, place, and context.

A task, from its creation to its destruction, goes through different states. The set of states is given in Fig. 2.4, along with the transition conditions. A task is represented by its task frame, which is a memory region. Thus the task – its task frame – is *owned* by a node at any time (see Chapter 3 for details).

Life of a task A newly created task is in the Created state. It stays in this state until the synchronization counter arrives to 0. A created task is owned by the node which called `task_create`, and *ownership does not change* while in this state. Thus `task_set_arg` requests must be forwarded to the owner node if it is called from remote nodes.

When a task has all its dependencies resolved (SC equal to 0), it moves to the Eligible

Figure 2.4: Life of task (possible states)



state. In this state the task could be executed (all arguments known), but we need to chose a node to run it, and potentially load some remote regions on the node. While in this state, the task can be moved between nodes for load balancing (*ownership can change*).

At some point, a node will decide to execute an eligible task, and move it to the Caching state. In this state (and all states that follow), the task is committed to this node and *ownership cannot change*. If the task has memory regions as arguments, the node must use the coherence system to retrieve remote regions locally before executing the task. The scheduler will send coherence requests upon entering the caching state, and wait for answers.

When all regions are locally available, the task goes to the Ready state. This state indicates that the task can be executed immediately (all arguments are locally available). It stays in this state until it is scheduled to run on one of the node cores.

When a task is scheduled on a core and starts running, it goes into the Running state. A task cannot be interrupted, so it will lock the core until completion. While executing, a task can use API calls like `task_create`, `task_set_arg` and `task_tdec`. `task_tdec` are registered and their execution delayed until the end of the next state.

At the end of execution, a task goes into the Publishing state. In this state, it must *publish* the changes made to the program state to every node. Changes include task state (arguments and SC) and modifications to regions (send invalidations in the coherence protocol). To ensure correctness, dependent tasks are not notified (`task_tdec`) until region modifications are published. After all publishing operations have been performed, the task is destroyed and the task frame deleted.

A formal presentation of this state transition system is available in Chapter 3 as part of the coherence protocol formalization.

Scheduling implementation The first role of a scheduling system is to execute tasks according to their dependencies. The formal correctness of this system is discussed

in Chapter 5. Intuitively, if a task enters the running state then all arguments are available locally (coherence), and have been computed beforehand (synchronization counter). Correctness requires that dependencies given by the user are correct, and that there is no *data race*. Incorrect dependencies might let a task run without all its argument computed. Data races (writes in parallel with others accesses) might trigger invalidations of the remote regions copies used by a running task.

The current design of Givy has two load-balancing steps. The first one operates between nodes: each node has a *queue of eligible tasks*, and can exchange eligible tasks with other nodes. The second one operates between the cores of a node: each core has a *queue of ready tasks*, and can exchange ready tasks with other cores. In both cases, the basic strategy used is *work stealing* [23], which consists of *stealing* a bunch of tasks from a random partner when our queue is empty. Work stealing is a simple yet effective strategy, and often outperforms more complex strategies.

Ready task load balancing can use shared memory work stealing, for which very efficient algorithm already exists. Eligible task load balancing implements work stealing by sending *steal requests* to other nodes.

In the context of Givy, eligible task load balancing can be smarter than pure work stealing. Using information from the coherence and allocator system, we can selectively push some tasks to other nodes:

- if we do not have fresh copies of the required remote regions, and know that another node has them
- if we do not have enough memory left to locally load the required regions (embedded context)
- have too much tasks in our queue (tasks that created a lot of new tasks)

We could also think of defining scheduling policies for tasks, for example to help spreading the computation evenly when creating a lot of new tasks.

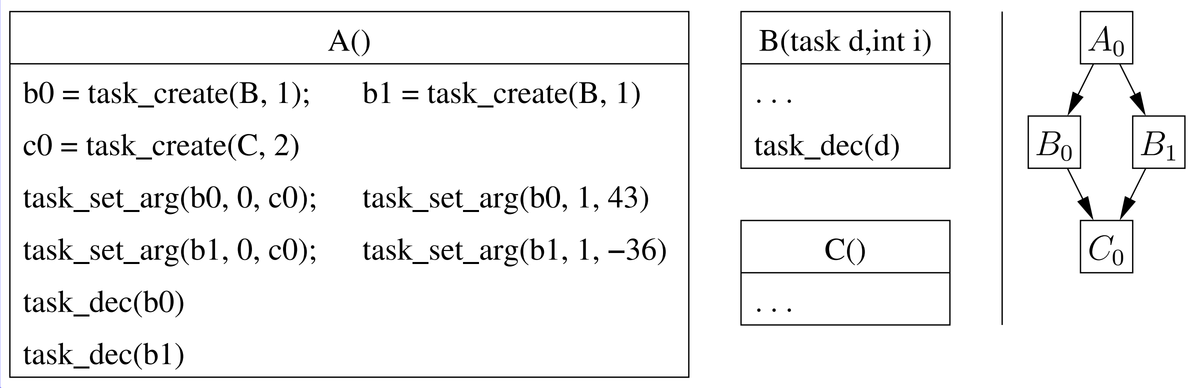
As an optimization, some of the tasks states can be bypassed if the task already fulfills the conditions. For example, a newly created task with all arguments already locally available can be run immediately. Bypassing queues is useful to reduce latency of execution, especially if few tasks are alive (and we need few cores to compute). However, if a sufficient number of tasks are waiting, some should be put into queues so that more nodes / cores can be used for the computation.

In our scheduling system, tasks (task frames) are not always referenced globally by the run-time. Instead their references are passed through the different structures until deletion:

Created	referenced by argument producer tasks (user)
Eligible	eligible node queues (runtime)
Caching	referenced by caching requests (runtime)
Ready	ready core queues (runtime)
Running	referenced by worker thread (runtime)
Publishing	referenced by publishing requests
Finished	deleted

No garbage collection of tasks is provided; a created task must be executed. Task cancellation is not provided yet. Cancelling a task could be done if it is still in created state, and all tasks referencing it are updated, but semantics of such operation would not be simple.

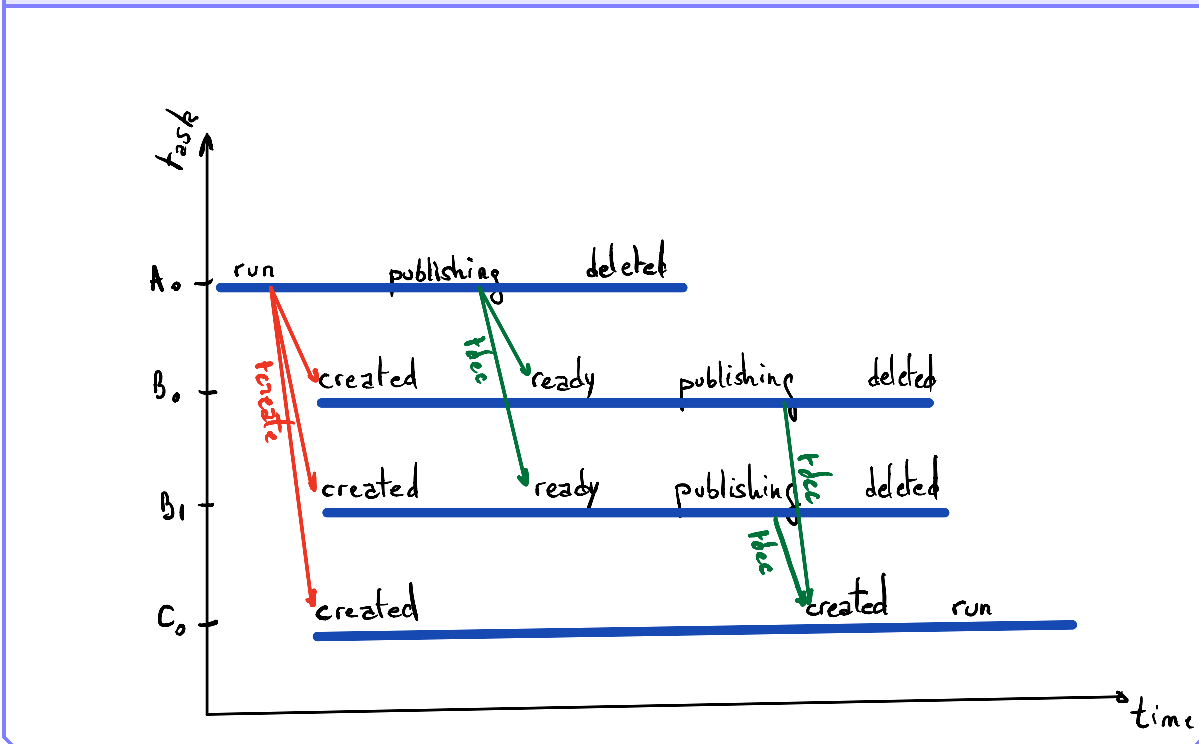
Figure 2.5: Example of a small task graph. `A`, `B` and `C` are task models (functions that describe tasks). $\mathtt{A}_0$, $\mathtt{B}_0$ / $\mathtt{B}_1$ are task instances created from their models.



Example Fig. 2.5 illustrates how to define a small task graph (on the right) using Givy runtime primitives (on the left). In this example, we will do a *fork-join* construct, which is creating parallel tasks doing the same job with different arguments, and then waiting for all to complete. Here task B is the operation to parallelism, with a dummy integer argument. Task A is the fork task; it will create all B tasks, gives them arguments, and uses a `task_tdec` to let them start computing. C task is the join, which does not actually do anything because the join will be handled by the run-time. We want to outline that to perform `task_tdec` on C, B tasks must have a reference to C. This is made possible as A creates both B0, B1 and C and can forward reference to C to B tasks.

Fig. 2.6 shows us some steps of a possible computation, starting with a A task created by an ancestor.

Figure 2.6: Example of execution trace (runtime events) for the small program in Fig. 2.5.



2.2.3 Allocator

The *memory allocator* component is used to manage the global address space. In the current Givy system, heap data is in the GAS, thus `malloc()` allocates blocks in the GAS, and the raw pointers it returns are GAS references. So the `malloc()` implementation is responsible for managing the GAS memory, both at a node-local level as a normal allocator, but also at the GAS level. The allocator subsystem of Givy is described in more details in Chapter 4.

The GAS is defined as an interval of addresses in virtual memory. Virtual memory allow us to use the same addresses on every node, assuming we create the right mappings. At any given time, each node is only required to map the part of the GAS that is locally used by tasks. Thus Givy can process a huge data set even if physical memories are small, by spreading computations (and data requirements) to multiple nodes: each node only needs to load a small part of the data set.

As classic memory allocators, the Givy allocator must manage the state of virtual memory. However it must coordinate its instances between nodes, to avoid *collisions* (allocating overlapping blocks). In Givy, this is done by giving a exclusive area of the GAS to each node (Section 4.2 and Fig. 4.2). Each node allocator will exclusively allocate and deallocate from its own area. This allocator also manages virtual memory associated to remote areas (areas given to other nodes).

The allocator subsystem works closely with the coherence subsystem (red arrows in Fig. 2.2). Management of the *remote areas* is driven by coherence requests (such as request to load a memory region allocated at another node). As `free()` is a GAS operation, it may trigger coherence requests to free copies on other nodes. Lastly, the

allocator is responsible for placing, and providing access to coherence meta-data for each memory region.

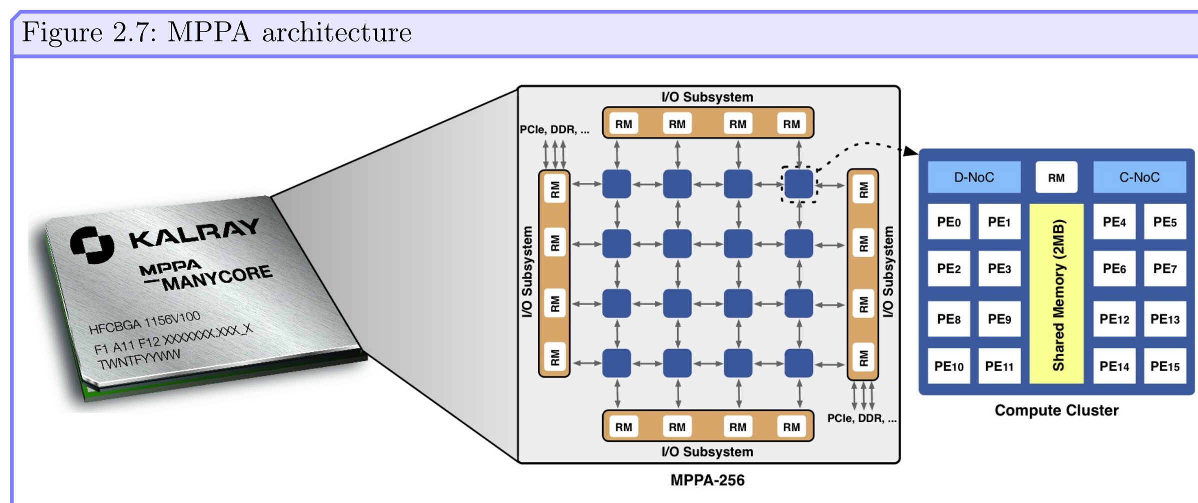
2.2.4 Coherence Protocol

The last critical component of Givy is the coherence protocol. It is a piece of event-driven code that maintains a coherence between the multiple node copies of a region. It relies on the memory allocator for managing the node-local address space. It requires meta-data to be associated to each region: this is also handled by the memory allocator. Coherence is achieved by exchanging network messages between nodes, which are described formally in Chapter 3.

2.3 Implementation

2.3.1 Version 1

Figure 2.7: MPPA architecture



Originally, Givy was targeted for the MPPA architecture (see Fig. 2.7), which was being developed at the Kalray company. This architecture contains 16 nodes of 16 cores each (and one additional management core), with everything integrated on a single chip. This target added two important constraints to the run-time: it needed to fit into the 2MB memory available on each core; using the C language as C++ support was not a priority.

Thus the first version of the run-time was written in C. This version was entirely composed of the version 1 of the Givy allocator (Section 4.4). The version 1 of the allocator was designed heavily around the small footprint constraint, with simple slower structures to reduce code and meta-data size. It did not handle the coherence meta-data placement; meta-data was placed inline with the data.

An interesting problem is to generate the task frame structures for each function. In C there is no clean way to achieve this, so those structures would have required an external generator (small C compiler, or letting the higher level run-time handle the generation).

The version 1 run-time (or just allocator) was never tested completely on the MPPA as software support for virtual memory manipulation was not available at that time. Only the cuckoo hash table of the Page Mapper Fixed that used MPPA-specific atomics (the MPPA memory model is weaker than the weakest C11 atomic model, *relaxed*) was tested on the MPPA. For the published paper [39], the run-time was tested on x86 using `mmap()`. Work stopped on the version 1 shortly after to give rise to the version 2 described below.

2.3.2 Version 2

The version 2 of Givy is a new implementation in C++. To speed up development, it was decided to work on x86 while keeping the MPPA in mind. C++ was chosen instead of C: the later version of MPPA added support to C++ compilation; also C++ allows greater refactoring and reuse of structures in case they needed to be scaled to the MPPA 2MB.

The current version 2 of the protocol contains the version 2 of the allocator (Section 4.5), a basic network layer, and a partial implementation of the coherence protocol. The version 2 of the allocator has a more modular and advanced structure, and can store meta-data on demand for regions.

Compared to the C version 1, in C++ we could generate the task frame types using advanced template constructions. While this is somewhat automatic and stays within the language (no external code generator), this would have lead to unreadable code and difficult maintenance.

An interesting problem is the virtual address placement of the GAS. The chosen virtual address interval must be available on all process taking part to the GAS. Currently Givy chooses a hard-coded address interval far in the address space, which seems to be free from interference even under *Address Space Layout Randomization* (ASLR). GASnet [24] tries multiple intervals using heuristics in order to reduce the risk of GAS initialization failure. It would ultimately be better to use similar strategies for GAS placement.

Listing 2.1: Interesting C++ constructions

```
constexpr T f (void) {...} // compile-time executable function
constexpr T v = f (); // compile-time generated value
T * p = new (addr) T (args...); // placement new
BoundUint<N> n; // uint8_t, uint16_t, uint32_t...
// Curiously Recurring Template Pattern (CRTP)
template<typename T> struct Link {
  Link * prev; Link * next;
  T & access_struct (void) { return static_cast<T&> (*this); }
};
class ChainableStruct : public Link<ChainableStruct> {
  int blah;
  ...
};
```

C++ features The C++ language is very large and offers a lot of different programming paradigms, with some having run-time overhead. Using C++14 with templates

and classes is surprisingly effective for low level programming, and offers some high level features without any overhead as most functions are inlined. Static (compile-time) polymorphism with templates allow to reuse structures commonly used like lists for chaining objects. Listing 2.1 gives multiple examples of useful features in C++14: Static arrays can be filled with values pre-computed at compile time (constexpr); Placement new allows to construct an object at a specific place (used to add a header at the start of a chunk of memory); `BoundUint` is a custom type that aliases to the smallest `uintK_t` that can contain the value `N`. CRTP allows types to be inserted in a linked list while being able to switch between link and object references.

Network layer Most networking library prefer a single threaded environment – API calls should come from one thread, or at least be sequentialised by a mutex. For example, only some of the MPI implementations support multi-threaded API calls.

In the current implementation of Givy, a *management thread* is created on each node. This thread is the only one with access to the network API. Each thread (worker or management) has a lock-free queue of events which acts as a *mailbox*. The management thread is in charge of:

- receiving coherence messages from the network, and reacting according to the coherence protocol (usually consist of updating coherence meta-data and/or sending an answer).
- sending network requests posted to its mailbox by worker threads (requests originating from `task_tdec`, `task_set_arg`, or caching requests).
- managing the eligible queue and eligible task work stealing messages.

2.3.3 Network

RDMA *Remote Direct Memory Access* [52] is the capability to perform some network transfers without involving the remote system user code. These one sided operations are initiated locally, and will be managed by the remote network card directly. API calls are usually similar to their classic message passing version, except for a *remote* memory address which is required to indicate where the operation will be performed on the remote node.

Using RDMA diminishes the need to interrupt computation to handle incoming messages. It also increases performance as the network card can transfer data in parallel to the computation performed by the processor.

However RDMA requires setup: *memory registration* (locking the mapping between virtual and physical memory) must be performed as most network cards manipulate physical memory and will have undefined behavior if the mapping changes during transfer. RDMA also does not always notify the remote node user code when a transfer has been performed. This type of notification is needed in a coherence protocol to indicate that a copy has finished, so we must still use regular message passing (for notifications) alongside RDMA for heavy transfers.

In Givy version 2, it was decided to use RDMA transfer only for large regions with an adjustable threshold, and use regular message passing for small regions.

Network API GASnet [24] is one of the most popular frameworks designed to manage a partitioned GAS (for example, used in Global Arrays [53]). It handles GAS setup (converging on a common GAS address between nodes), and RDMA setup (registration) for various HPC network systems. However the whole GAS is registered statically for the duration of the computation. Most systems have a limitation on the amount of registered memory. So GASnet does not fit well with our approach where the GAS might be large (event if few objects are actually transferred).

We tried to use the CCI [18] network library, which aims for high performance and low size. CCI might be ported to the MPPA, so it would allow Givy to support the MPPA without major rewriting (of the network layer at least). However, CCI is not adapted to the irregular point-to-point communications Givy can generate. It requires to create explicit *connection* objects for each path you might take, which is quadratic in the number of nodes and thus not practical. In addition it creates an internal CCI management thread for common network back-ends. We cannot integrate our management thread with CCI's one, so we end up with two management threads which is quite wasteful.

In the end Givy version 2 is built upon MPI [37] which allows irregular point-to-point operations. MPI also sets up node numbers during initialization (another feature that must be re-implemented upon CCI). RDMA is available. Memory registration is dynamic which fits well with Givy if we want to use RDMA.

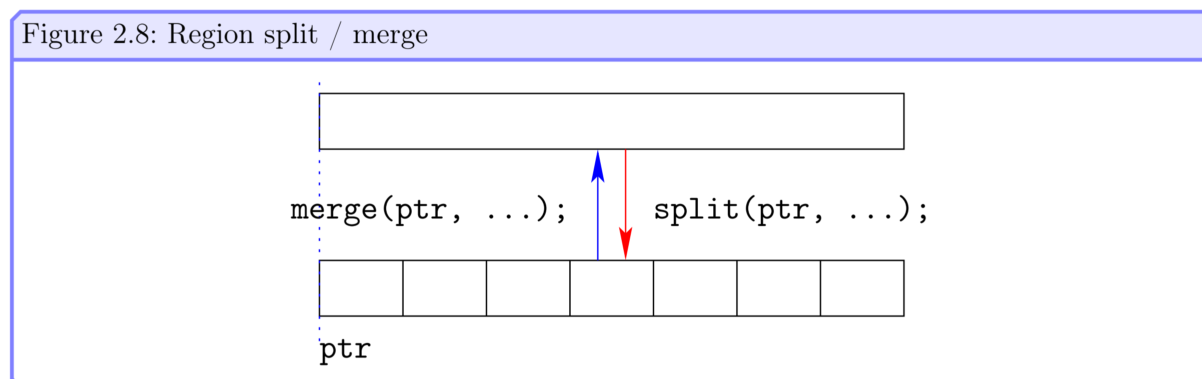
For the MPPA, Kalray has developed `libnoc` which is a low level API to use the MPPA network system. Currently Givy is restricted to x86. `libnoc` is probably the best network API option for a port to the MPPA (less overhead, highest probability of being able to integrate management code in the *resource management* core of each node).

2.4 Extensions

This section presents possible extensions to the current Givy run-time system. They have not been implemented, but some (especially split / merge and mutex) are important. Some extensions reference details of the coherence system, so it might be useful to read the Chapter 3 first.

2.4.1 Region Split / Merge

Figure 2.8: Region split / merge



The current Givy task and coherence model is relatively strict and can be incompatible

with some common patterns. Let us suppose we want to apply a function `f` on all elements of an array and build the array of results (a map operation). Also suppose we build a task for each call of `f`. Theses tasks can all access the initial array in parallel without problem, as it is a read only access (no data-race). However accesses to the result array must be serialized because they are writes.

We propose a solution to this problem with the split and merge operators in Fig. 2.8. From an initial region and a cutting pattern, split creates N *sub-regions* which represents exclusive parts of the initial region. The merge operator takes a pointer to a region split into sub-regions and restores the initial region.

The split operation only influences coherence meta-data. On a split, an array of meta-data must be created which will represent the meta-data of each sub-region, and the meta-data of the initial region is flagged as split. As the split operation has write access, all other node copies will be invalidated. So all other nodes will fetch the new meta-data state when trying to access the region, and be aware of its new split state. The initial region should not be used until a merge is performed (and meta-data returns to its initial state).

The split / merge operations solve the initial problem of a map: we split the buffer into cells, write to them in parallel (no data-race as they are distinct regions), then merge the resulting buffer.

2.4.2 Region Set

A second extension is to have a special argument type that represent a set of regions. In the caching state, all regions of the set will be loaded instead of just one per argument. This is a possible way to implement usable pointer-based structures, however we still recommend against generic pointer-based structures in Givy.

A better way to implement a pointer based structure (like linked lists) is to use a memory pool that is used as a splittable region. This way, all elements of the structure are part of the same global region, allowing a fast access to the complete list. And it also supports splitting the pool in sub-regions, for parallel accesses. As a bonus, it force some memory locality for the structure, which might improve cache performance.

2.4.3 Region Mutex

Another common computing pattern is an un-ordered but serialized access to a region. This pattern is useful in the case of region indirect access. If the choice of which region will be accessed depends on data, it can be impossible to guarantee that no data race occur (two modifying parallel accesses to the same region).

A solution is to have an optional *mutex* mode for regions. In this mode, a region that is currently in use by a task is locked Any other task wanting to access this region must wait for the current task completion.

This mode has semantics similar to adding *mutex* on the region. Thus all problems associated with mutexes can appear, like a deadlock between tasks. However, as the mutexes on task required regions must be taken by the run-time, schemes designed to avoid deadlocks can be used. This mutex mode also adds some overhead, which is why it should be optional (toggled in a way similar to the split/merge system).

2.4.4 Stack and Globals Support

In the current Givy, the GAS is restricted to heap space. This means that other storage categories can not be part of the GAS: stack variables, global variables, and `thread_local` variables. Adding support to these variables has the same requirements than for the heap: Meta-data must be associated to each variable. Stack, global variables (`.data / .rodata` sections), and `thread_local` variables must be placed in the GAS node area.

The memory address placement can be controlled but requires to override OS settings. The Stack can be relocated at program startup to a reserved part of the node area. Global and `thread_local` variables placement requires internal OS support to be placed in a GAS reserved area.

Lifetimes of regions must be known to be able to manage meta-data (create / reserve meta-data at region creation, cleanup at region destruction). This feature is more difficult to achieve than for the heap: Givy does not have control of the layout of objects like in the heap. And it does not have a natural instrumentation of object creation and destruction like with `malloc()` and `free()`. Such instrumentation could be added by using compiler support. Another way, using C++, is to use the constructor and destructor of classes. However it restricts GAS support to only class objects, and requires adding inheritance to some Givy instrumentation class (intrusive).

To conclude, support of anything other than the heap goes must use more than a library support. It must be intrusive, and use OS / linker / compiler support to provide the necessary properties on non-heap variables for Givy.

2.4.5 Code Section Support

A topic which has not been discussed is how tasks reference their code (the function that implements the task functionality). In the general case there is no guarantee that a function pointer stays the same between different processes. In practice this is often not the case due to dynamic linking, or *Address Space Layout Randomization* (ASLR) used for security in most x86 OS.

On the initial MPPA target, this was not a problem as the specific linker configuration ensured the use of the same code address on every node. Thus function pointers are valid globally, and can be used to reference function code in task frames without problem.

On the more generic x86 target for Givy v2, function pointers are not global anymore. A first possible solution is to index all functions used as task bodies and use the index as a global reference. A table must be built on each node to associate the local function pointer to the global index. The table and indexes can be generated by either a small compiler pass, or by the higher level run-time using Givy as a back-end. GASnet [24], a popular framework to develop GAS systems, uses this strategy. The second possible solution is to place the code section (`.text`) in the GAS, and requires OS support.

2.5 State of the Art

In this section we look at other DSM which are interesting with respect to Givy.

Treadmarks Treadmarks [15] is one of the first DSM systems. It is a software NUMA: it has a threaded execution model, uses real pointers, and operates a page granularity. Coherence mechanisms are triggered by page faults (retrieves using signal handlers). In an effort to reduce false-sharing (due to the page granularity), it exchanges diffs of pages instead of whole pages. Due to its execution model, it does not balance computation between nodes, and has blocking coherence mechanisms.

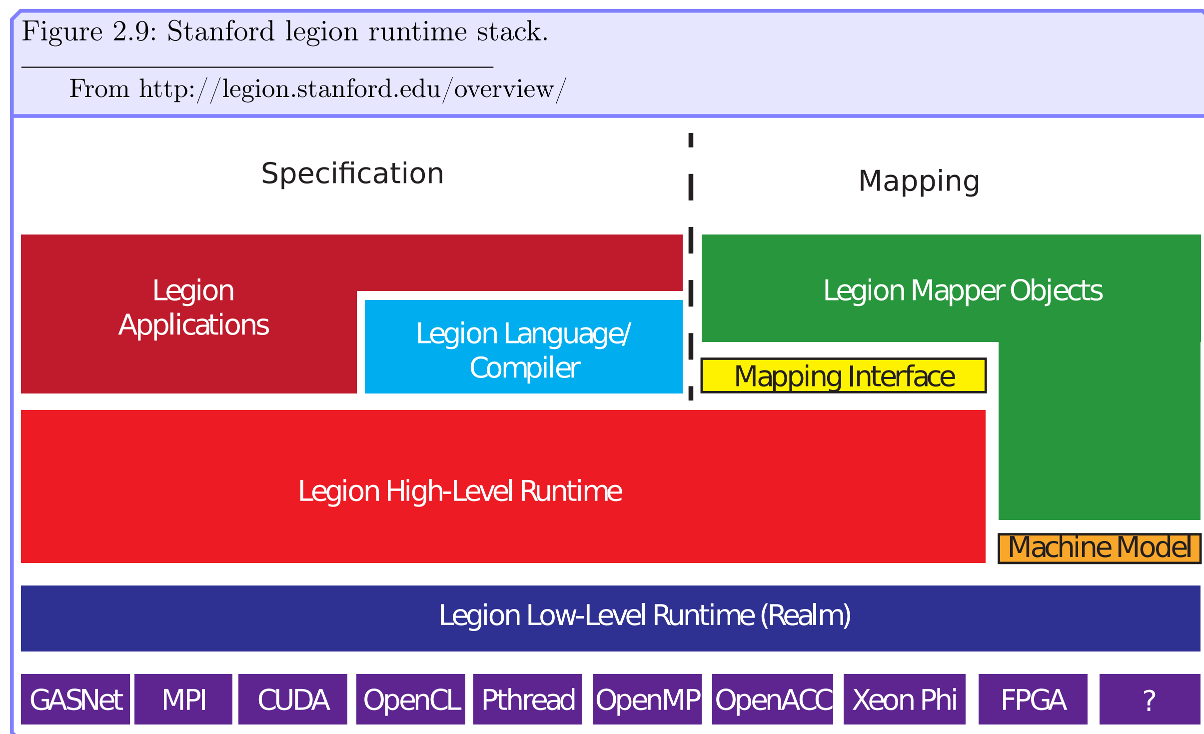


Figure 2.9: Stanford legion runtime stack.

From http://legion.stanford.edu/overview/

Legion Stanford Legion [20] and its task runtime Realm [69] is a more recent DSM. The legion DSM is quite flexible and proposes both a C++ API, and a DSL (*Domain Specific Language*) interface built on top of it (Fig. 2.9). Legion is a task-based, library, fake-pointer / index based runtime. Data can be replicated between nodes, and is indexed through either fake pointers `ptr_t` or keys in tables. Data initialization can be used to set a initial partition across nodes. Tasks live together with *events*, objects with dependencies which can be triggered. Events allow to represent and organize complex dependence schemes between tasks and other action objects. Legion provides a GAS (based internally on Gasnet [24]) but does not provide automatic coherence. Special *copy* actions allow to transfer one version of a global object from a node to another.

Intel CnC Intel *Concurrent Collections* (CnC) [25] is another task runtime with a C++ API. It is a task-based, library, index runtime. Computation and data placement is completely abstracted away in GAS *collections*. These global sets indexed by tags give access to value, or define the set of tasks to execute. A task body can request values from value collections, which will trigger GAS mechanisms to retrieve the data. Instead of requiring the application to be data-race free, a value in a CnC collection can only be

set once (no update). The collection placement of data and tasks can be tuned to achieve better performance.

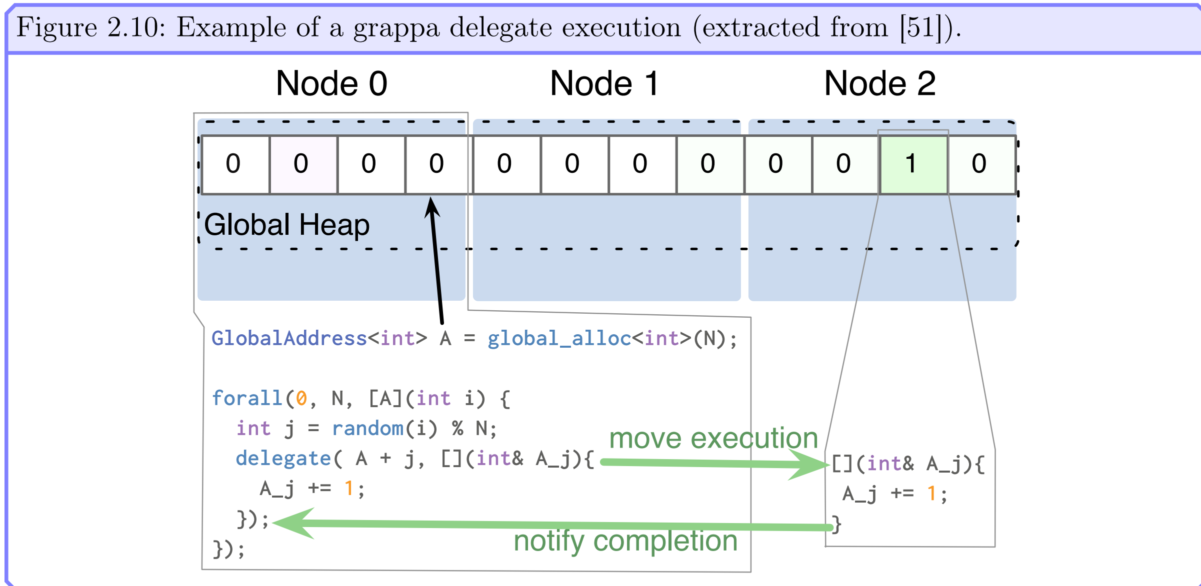


Figure 2.10: Example of a grappa delegate execution (extracted from [51]).

Grappa Grappa [50, 51] is a modern PGAS task runtime, with a C++ API. In Grappa, memory blocs of nodes can be registered in a GAS (with a fake pointer). User code can then create tasks called *delegates*, linked to a GAS reference, which will be executed on the home node of the GAS object (Fig. 2.10). These remote delegates are scheduled with currently running on a node using fast context switch. Grappa simplifies using RDMA network capabilities with a GAS. Memory coherence is achieved by serializing on the home node all tasks and delegates that access a specific piece of data.

HPX High Performance ParallelX (HPX) [41] is a task GAS runtime with a C++ API. The GAS system backing HPX, called AGAS (Active GAS), manages blocks referenced by fake pointers (`hpx_addr_t`). Blocks can be accessed globally through RDMA put / get calls, or by using *parcels* (small functions to be executed on the home node of an associated block). In HPX case, the blocks can be relocated between nodes, after an initial distribution. Control flow between tasks is defined by *local control objects*, which implements common patterns like reduction, semaphore, futures.

Myrmics Myrmics is also a task based runtime for distributed architectures. It is described in an overview arXiv paper [45], a PhD thesis [44] and an early ISMM paper describing the memory allocator system [46]. Tasks are declared by using `#pragma` annotations in C code, which are used by a source to source compiler to generates calls to a runtime API. Myrmics provides a true global address space using real pointers like Givy. However these pointers must be attached to *regions*, which are GAS memory pools, and allocated into them using a specific API. Task arguments are scalar values, single GAS objects, or regions (requests all region objects). Regions are organised hierarchically as a tree, and assigned to *schedulers* (nodes dedicated to runtime management, as opposed to worker nodes). Tasks management is also assigned to a fixed scheduler at creation, and

child tasks spawn on either this scheduler, or one of its children scheduler if any exists. Schedulers are also organised in a tree that should map to the architecture, and communications follow the scheduler hierarchy (instead of flat all-to-all communications). The hierarchy of memory layout (regions) and computing (tasks) is used to provide locality that can be used by the hierarchy of schedulers, which can then map to the parallelism and locality hierarchy on the target platform.

Task dependencies are determined by the order of task spawn operations. Each region and object has a *dependency queue* representing tasks awaiting access to the region or its child region / objects. On each task spawn, the task is added to dependency queues of the topmost regions of its parent, if they request access to these regions or one of their children. When a task is at a bottom of dependency queue for a region but only uses children regions / objects, they are moved to their queues. When a task is the bottom of all dependency queues of objects they request, it is ready for execution. It is assigned to a worker node, and the required regions are transferred from the last worker nodes that wrote to them.

GAS layout management and memory allocations are also managed hierarchically. The root scheduler owns the whole address space. Each scheduler requests new space for regions from its parent scheduler when it cannot serve the request itself. Each region is associated to a *segregated slab allocator* (Section 4.1.3): this allows region objects to be tightly packed in memory for bulk transfers.

Myrmics has many similarities with Givy, mostly due to the choice of real pointers. Givy is organized with a fixed depth hierarchy (nodes and threads), compared to the extendable hierarchy of Myrmics. Givy memory allocation is mostly free of network transfers (only `free()` generates a non blocking send). Givy hides memory organization from the programmer, as it does not ask the programmer to organize GAS objects in a region tree. The bulk dependency specification that region provide can be done by either region splitting or region sets in Givy. Region splitting are effective for array data (matrices), but requires explicit transitions between the splitted and complete version of the memory block. Region sets seems less practical that the Myrmics region scheme however. Task dependencies are deduced from spawn order in Myrmics, which lessens programmer burden. Givy dependencies are explicit but allow more irregular patterns, at greater programming cost.

The Myrmics region system as also been used as a standalone library to simplify MPI programs, called DRASync [66].

STAPL STAPL [16, 8] is a C++ parallel programming framework supporting both shared and distributed memory systems. It provides *pContainers*, which are data structures with a similar API to STL containers (vector, list, etc). Internally, pContainers distribute their data across the distributed memories and maintain metadata on the physical locations of fragments.

STAPL provides *pViews*, the equivalent of STL iterators for pContainers. pViews are mostly used to define data ranges on which computation will happen. They can also be used for single element access. Each pView can be partitioned into subviews, which is used to adapt computation granularity to the target machine level of parallelism.

pAlgorithms are parallel equivalents to the STL algorithm library. Most STL algorithms (`std::find`, etc) have a STAPL equivalent taking pViews instead of STL iterators.

These algorithms are then executed in parallel over the distributed system, by choosing an efficient implementation for the target machine. Internally, pAlgorithms are encoded as task graphs (*PARAGRAPH*). Basic usage of pAlgorithms seems limited to a SPMD model, with a main controlling thread using pAlgorithms sequentially.

Advanced users can define their own pAlgorithms using the *Skeleton* framework [71]. A skeleton is a parametric data-flow task graph, defined by composition of basic operations (map, reduce, etc). A task graph is then instantiated based on the skeleton and the computation data (pViews), which will determine the task graph dimensions. An application defined as a composition of skeleton can avoid the global synchronization of sequences of pAlgorithms.

STAPL-RTS [59] is the STAPL runtime system. It provides basic facilities like threading, performance measurement (for pAlgorithms tuning), network transfers. Communication primitives are remote invocation of functions, called *Remote Method Invocation*, similar to GASNET active messages. It also provides a scheduler to execute task graphs.

Compared to Givy, STAPL has been developed with a big focus on its interface. The similarity with the well known STL containers, iterators and algorithms can greatly help conversion of existing codebases to STAPL. This interface is used to generate instances of task graphs that are then executed on the distributed system. Givy has no such interface, and users are expected to encode their program as tasks manually for now, which is a huge usability problem.

STAPL provides a logical global address space with the pContainers. Each pContainer must define how its data is distributed internally, and how to maintain the coherence between fragments. Givy put the entire heap in the global address space, which is more general. This means Givy can be used with existing data structures, as long as all access to them are annotated (tasks). However STAPL has more high level information to efficiently distribute data.

As STAPL has no pointer level GAS, each data transfer must serialize the transferred data. User defined types must provide serialization primitives, which can be complex in C++ with inheritance. Givy does not need serialization nor object serialization primitives: it can transfer the object binary representation directly using RMA. However this restricts Givy to distributed systems with homogeneous architecture (type sizes, etc).

Chapter 3

Coherence Protocol

A *Cache Coherence Protocol* is a tool to synchronize data between distributed storage units. It is defined by: a set of *network messages*, *metadata* attached to piece of stored data, and *rules* that describe how to interact with memory accesses (from the processor) and network. Typically, copies of the piece of data will be stored on some storage units, with metadata describing if they are *valid* or not. Network messages are used to inform other storage units of data change, retrieve new data locally, and so on. The protocol goal is to maintain a *coherent* view of data across all units in the system.

Cache coherence protocols are not new. They have been with us since the advent of multicore processors, maintaining coherent data between the processor core data caches. These protocols are called *hardware cache coherence protocols*. They are usually implemented in hardware, and exchange message on an internal processor bus. Storage units in this case are the data cache, which store copies of data from the main memory. Memory accesses are *load* and *store* instructions, with the addition of special instructions like *flush* and *CAS*.

Givy – and some other DSM systems – use a *software cache coherence protocol*. Storage units are node local memories. Protocol messages are exchanged on the network liking the nodes. This protocol is called *software* because it is usually implemented in software (except in hardware NUMA machines which have hardware implementations). Even if the environment is different, the concept and goal are the same, and the same kind of rules can be used.

3.1 Formalization Flavors

3.1.1 Operational

A simple way to describe a coherence protocol is to describe all mechanisms involved in the protocol itself. This representation is called *operational*.

It consists of describing a *system state* and *transitions* (rules) that can occur and change the state. The state describes the entire system at a given time. It can be decomposed into smaller states representing the various parts composing the entire memory system, which are assembled together to form the global state. The transition system is a set of rules. Each rule is composed of a predicate over the global state (guard), and a modification of the global state. To simulate the system, at each step a rule with a

true predicate is chosen, and the corresponding state modifications are applied. Such a description is very close to implementation code: the transition system is in fact a pseudo-code writing of the protocol.

For hardware coherence protocols, this type of representation describes the hardware design of the coherence subsystem. An example of an operational memory model for the x86 cache architecture can be found in this paper [57].

The Givy coherence protocol is described as an operational model. Operational models are precise, as they describe every element of the system: it enables simulations of the system. Moreover one of the initial goals was to potentially automatically generate the C/C++ coherence code from the model: it requires details only found in operational models.

3.1.2 Axiomatic

The second kind of model is called *axiomatic*. It has not been used for Givy, but it is still interesting as it is used in literature to describe the C++11 model used in Section 5.2. Contrary to operational models, axiomatic models describe *behavior* instead of actual mechanisms that implement the behavior. They are close to a system specification, and are useful when the system implementation is not know but its behavior can be tested (*black box*).

A good example of an axiomatic model can be found in a paper of Jade Alglave [13], with further details in her thesis [12]. An axiomatic model describes the behavior of the memory system. It tells which operations are allowed, and which ones are disallowed, but does not give details on why. It is usually presented as a function which takes an *execution trace* as input, and returns a boolean value telling whether this trace can happen according to the rules of the model.

An example program (Fig. 3.1) is written as a Litmus test [14]. Litmus is a hardware testing tool that takes a list of programs (here, P0 and P1), an initial state of memory (Init), and a boolean predicate (Observed). It then runs those programs in parallel, and after completion checks the value of the predicate. Such tests are repeated many times, to maximize the probability to see all possible outcomes when programs have data races (and thus have a non-deterministic behavior). Litmus allows us to capture hardware allowed outcomes, even if the hardware *implementation* is unknown (which is often the case).

x and y are memory locations, and r_1 and r_2 local registers. (a) and (c) correspond to write accesses, and (b) and (d) to read accesses. There are no synchronization between (a) and (d) (accessing x), and (b) and (c) (accessing y), so this program has *data races*. Assuming the instructions will be interleaved, expected outcomes at execution end will be either r_1 or r_2 set to 1, and the other set to 0 (this choice is non-deterministic). Modern processor can *reorder* instructions for performance (*out-of-order* or *superscalar* processors. This reordering can produce new outcomes, which is what is being tested in this Litmus program by the *Observed* predicate.

We now want to determine allowed outcomes of the model, and compare it to the Litmus results. More specifically we want to check if the model allows us to obtain the outcome of the *Observed* predicate. To do so, we need to build an *execution trace graph* that represents an execution of the programs which satisfies the predicate.

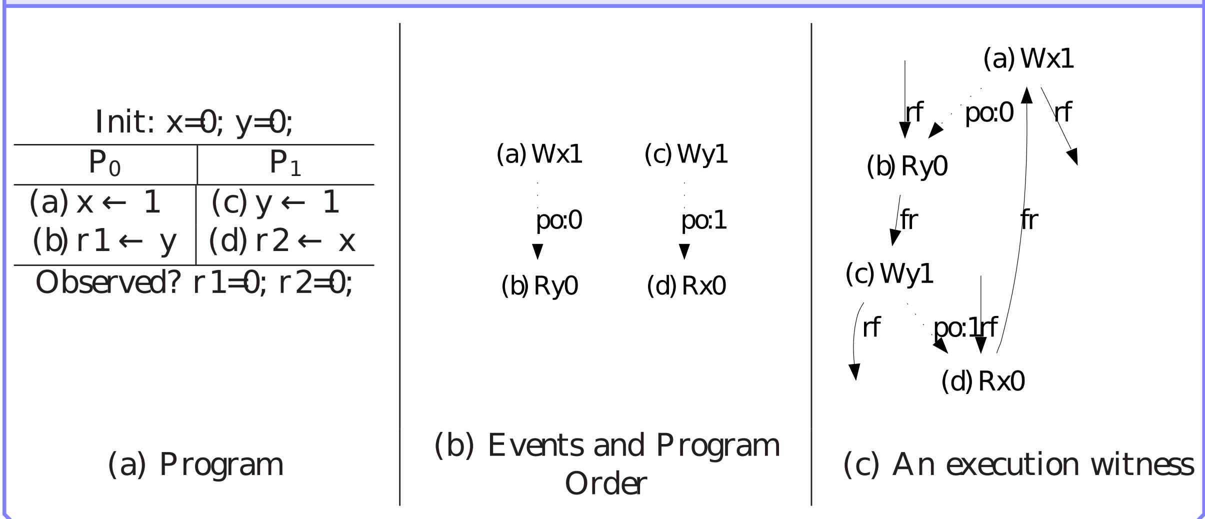


Figure 3.1: Example program and execution trace.

We first create a memory event for each executed instruction. Those events are the nodes of the execution trace, and can be seen in Fig. 3.1(b). The name of the event indicates the type of event (read or write), the storage location used (x or y here), and the accessed value.

We add dependence edges to this this graph. They define an ordering of memory events which should be coherent with the predicate result. The resulting *execution trace graph* is visible in Fig. 3.1(c) where: *po*-edges represent initial program order; *rf*-edges represent *read-from* (or flow-) dependencies (a read event reads from a certain write event); *fr*-edges represent *from-read* (or anti-) dependencies (a write event overwrites a value after a read event reads it).

To check if this trace is allowed by our model, we must check if we can find a total order on memory events that is coherent with the various dependencies. This order is usually called *happens-before*. In the Fig. 3.1(c) graph there is a dependency loop: thus no total order of events can be found as it would break some dependencies of the loop. Such a model, which takes into account all dependencies including program order is said to be *sequentially consistent*.

Superscalar processors and modern caches are however said to be *relaxed*: they do not conserve all types of dependencies. Processors must keep data dependencies, but are free to change program order is there is no data dependency. For such a processor, program order edges are ignored, and we can remove them from the graph. In the Fig. 3.1(c) graph, a total order can be found if we remove program order edges: the *Observed* predicate outcome is allowed by our relaxed processor model. Thus a memory system model is defined as a *mask* (filter) over dependency edges in traces: determine which types of dependencies are preserved by the architecture.

By comparing allowed outcomes from the model rules and allowed outcomes observed from the hardware, it is possible to check if the model and hardware behavior match. Given a set of test programs, a common strategy to model an unknown processor cache design is to iterate on the model rules until they match the behavior observed on the hardware. This experimental approach was used by Jade Alglave to model architectures like PowerPC. Real models use many more edge types than what was presented here, as

architectures are quite complex.

3.2 Cache Coherence Protocols 101

This section presents various classic coherence protocols. They are presented in an operational form (automaton with transitions as a graph). They are cache coherence protocols for multicore processors, found in this book [30]. As said before, the concept is general and applies for Givy as well.

Coherency A formal definition of *coherency* is given in this book [30]: a multi-processor system is said to be *coherent* if for any trace of any program, for every memory location we can find a global order of all memory events on that location (from all processors) which respects the following properties:

- events from a particular processor should be in order,
- each read get its value from the last performed write.

The second property can be decomposed into two important properties that directly impact cache coherence protocols design:

- *write propagation*: writes from a processor becomes visible to other processors,
- *write serialization*: all writes to a location are ordered and seen in the same order by each processor.

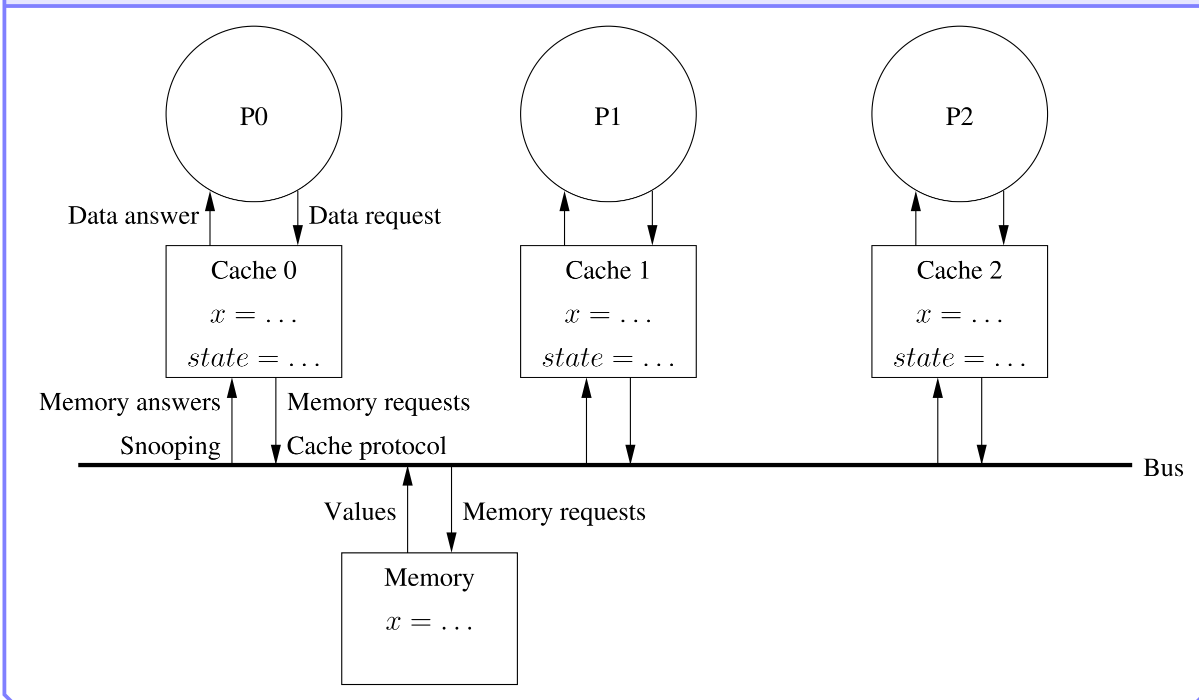
Architecture All following coherence protocol will be modeled for a multi-core processor architecture. Fig. 3.2 illustrates the architecture structure as seen by the models. The models only describe the protocol between caches: processor (and programs) are black boxes that emit memory accesses. Each processor (Pi) emits requests which are reads or write requests (for a specific memory location, and a value to write for a write request). Then they wait for the request to complete, and move on to the next request (issued by the next instruction).

In the models, each processor has a dedicated local cache. For the sake of simplicity we will consider one level of cache, and only one memory location (named x). With multiple memory locations, the request is just forwarded to the cache handling the memory location in the request. The cache can store the memory location value, and has a *state value* indicating how the stored value should be handled.

In these models, we assume caches will communicate between them and with the memory by using a bus. A bus is a flattened network model, where only one people can emit at a time, and all people can listen (which is called *snooping*). The bus is used for two things:

- interacting with the main memory: caches with an old value will ask for an up-to-date one,
- cache protocol messages between caches: when a cache signals others that it has written a new value for example.

Figure 3.2: Models architecture.



The following models will each time describe the cache internal behavior and the protocol of communication between them.

3.2.1 VI

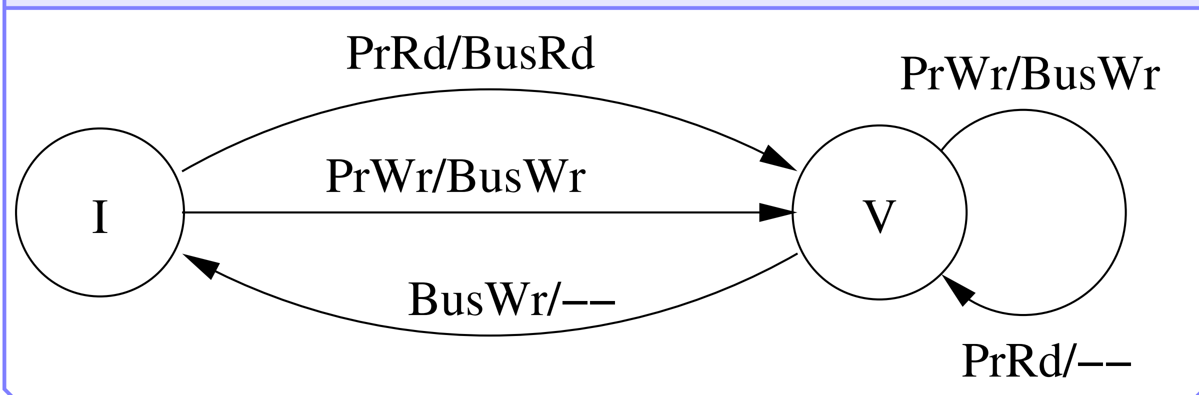
We start with the simplest cache protocol, named *VI* for *Valid* / *Invalid* (names of states). Data stored in the cache is tagged by a boolean value (*valid* boolean flag). This flag indicates if the data is up-to-date, or should be fetched from the main memory.

The cache protocol itself can be presented as an automaton, as shown in Fig. 3.3. Each state is a value of the valid boolean flag. The cache can react to events from processor and from others caches (through the bus), and change of state accordingly. The notation eventA/eventB means that if eventA is seen, then we must take the corresponding edge, and launch eventB on the bus. When eventB completes, we can go in the new state, and answer to the processor if eventA was a processor related event. If no edge belongs to the incoming event, this event is ignored.

The automaton has the following events: PrRd is a read request from the processor, and PrWr a write request. BusRd is a bus event saying that a processor want to read a value from memory. BusWr is also a bus event, which says that we want to write a value to memory. The two bus events complete when the memory executes the write, or answer with the value. These two events are supposed atomic (for one memory location): the bus should only allow one event at each time.

The automaton works as follows, as seen from a single cache line: Assuming we start in state I (*invalid*) on one processor which means invalid data. If the processor requests a read: we first ask data from the memory, and go into state V upon receiving it, as we have an up-to-date copy of the memory. This step is called *caching*. We can then answer

Figure 3.3: VI protocol transition system.



to the processor request. If it requests a write: we ask the memory to write our new value, and also move to V state upon completion (we stored the new value in our cache). We then answer the processor request.

Assuming we are in V state (*valid*), and have a read request: we can return the value to the processor as it is up-to-date. On a write request: we also need to send a bus request to memory, and wait for completion. If anyone else tries to write by sending a BusWrite request, we go to I state as our data will be out of date when this request completes. This is called *invalidation*.

This protocol ensures write-propagation, as a write will force other caches to go to I state, and to ask memory for the new value in case of a read. It also ensures write-serialization, because the bus will only allow one request at a time. Thus we have cache coherence.

This protocol is the basis of most cache coherence protocols, which often are extended version of VI. However it has a major problem: all writes from a processor will generate bus events, and take bus bandwidth. So this cache is not able to speed up processor-local data structures which are accessed in read and write modes, which is a very common occurrence. This leads us to the most common protocol, named MSI which can handle this pattern more efficiently.

3.2.2 MSI - Snooping

The VI protocol is a *write through* cache, meaning each write is propagated to memory immediately. To allow local read and write operations without modifying memory, we must use a *write-back* strategy. Writes are not propagated immediately but propagated when needed, as when another processor request the data.

Starting with the VI protocol, the V state is split in two new states, M and S (see Fig. 3.4). The S state (*shared*) represents a read only valid value, which is coherent with the main memory. The M state (*modified*) represents a modified value, which is not yet propagated to memory. This new state allows modifications to stay local until anyone else wants to read. To ensure write serialization, we must guarantee that only one processor can write, even it is not yet visible to others. So we must ensure only one processor is in the M state anytime.

We describe the MSI automaton of Fig. 3.4: It uses the same presentation as the VI

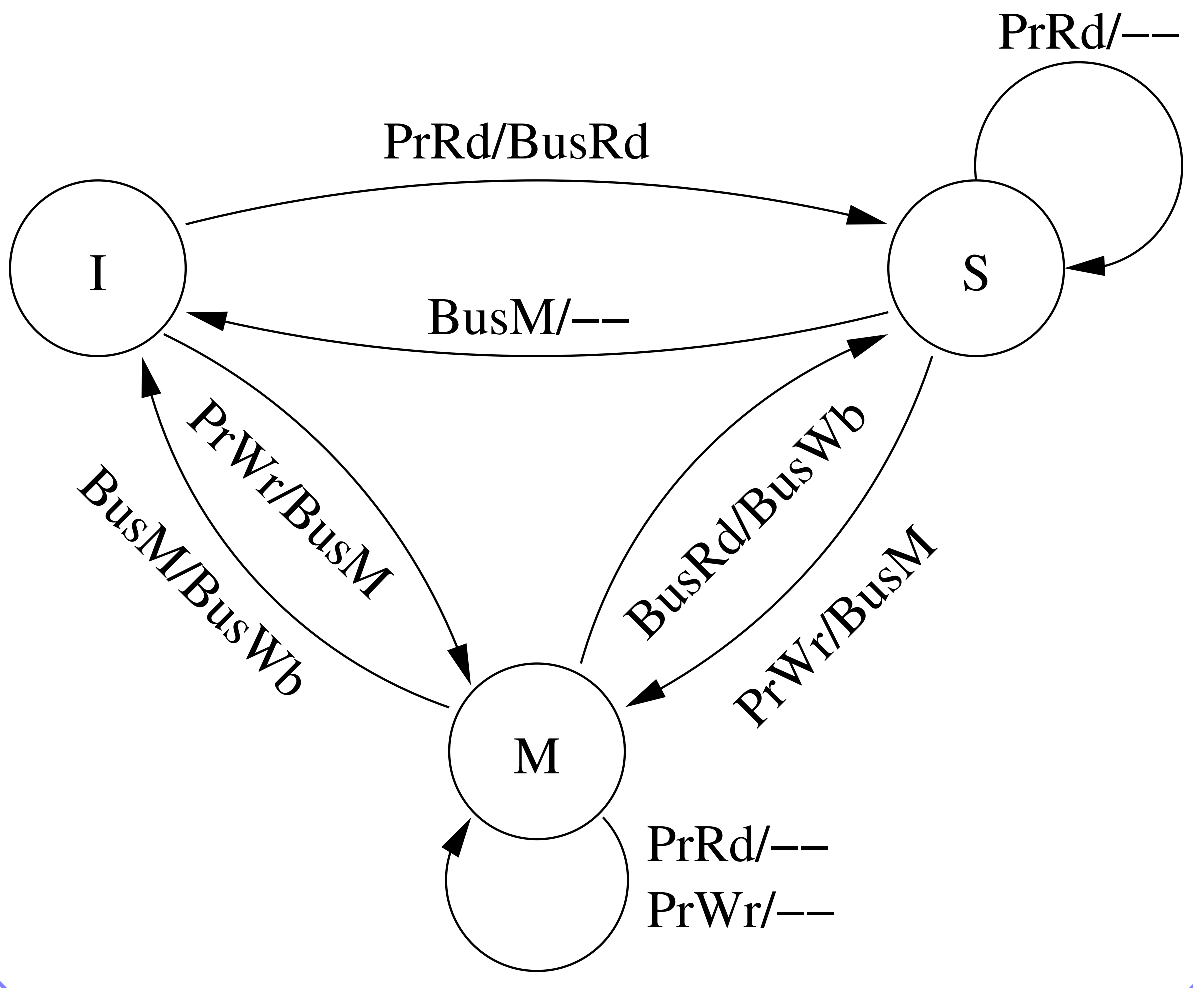


Figure 3.4: MSI protocol transition system.

automaton, and share most of its events. There are two new events, BusM and BusWb. BusWb is used by a cache to write its modified data to memory (*write-back*). BusM is used by a cache to signal other caches that it wants to go into M state.

The transitions are similar to the VI automaton. If we are in I state, we can either try to read, using the same edge as in VI, and go to S state as we just cached the value in read only. If we want to write, we must request to go in M state using BusM.

In the S state, we can read immediately, and we can write by going into the M state by also using the BusM event. If we see a BusM event, it means someone has made a write (which is not propagated in memory, though). So we go into I state to ensure fresh data will be loaded from memory at the next read.

In the M state we can read or write immediately, as we have the only up-to-date version of data. And it can be done without triggering bus events. We must write our modified version when someone tries to read (BusRd), and go to the S state (read, which means our data will be still up-to-date). Remarks that this operation will block the processor trying to read until the data has been written back. We must do the same when someone tries to write (BusM), and go to the I state as our data is not up-to-date anymore.

3.2.3 MSI - Directory

Those two previously described protocols were *snooping* protocols, meaning all caches listen to all events. For each state some events are ignored, and the protocol will still works if we do not send these events to caches in this state. Thus an optimization is to not send these ignored events: it reduces the amount of messages.

The main memory must store additional metadata, to track the state of each cache. Using this information, it can avoid sending ignored messages depending on the receiver tracked state. This is called a *directory* protocol. It can reduce the number of messages on the bus very efficiently, but it adds complexity and memory usage to the protocol. This reduction of bandwidth requirements can be critical if the network between the caches is not a bus, but a point-to-point network (where broadcast communication emulating a bus are costly).

3.2.4 MSI - Distributed

Finally, those protocols often target low-level systems, like one multi-processor system, where we have a main memory. But it can be used on high level multi processor systems, like complete machines over a local network. In this case, there is no main memory to rely on, only distributed memory, and we want to emulate a main memory with local caches.

One solution is to adapt the MSI protocol to always have exactly one cache in M state, and send all requests that were sent to memory to this cache. The coherence protocol of Givy falls into this category.

3.3 Givy Coherence Protocol

The Givy cache coherence protocol is a flavor of *distributed MSI* called *Owner Writable Memory*. It is a software protocol. As described in the runtime chapter (Chapter 2), this protocol operates on *regions*: dynamically allocated blocks of memory, of variable size. These regions are passed by reference (C pointers) to tasks (in their arguments). They are detected by the runtime which generates coherence protocol requests and wait for answers before executing the task. A read request is emitted if it is a `const T*` pointer. A write request is emitted if it is a `T*` non-const pointer.

When using regions task arguments, a strong property is required: no region level *data race*. Data races are defined by a write access in parallel with any other access. This means that if a task is accessing a region in write mode, all other tasks accessing the same region must have a dependency chain from or toward the write task. As tasks are atomic, we can consider them as a big memory access that access their all their regions arguments in their specified modes at the same time. This *data race free* property simplifies the protocol design. This loses some expressiveness, which can be restored by providing extensions for useful and controlled data races (like mutex-like behavior).

High Level Overview This small sections gives a high level overview of the protocol features. The Givy protocol is a distributed MSI: there is no main memory. Storage units are cluster node memories, referred as *nodes*.

For each region, the protocol will ensure that there is a node that *owns* it: the *owner* node. This node has the most recent version of the region, and is responsible for giving copies of this data to nodes wanting to read it. When someone else wants to modify the region, it must become the *owner* first, and then modify the region. This is a bit similar to the M state of MSI, but different as this M state must be persistent (we always need to have an owner). This is also why this protocol is also called *Owner Writable Memory* (OWM).

The protocol is *directory*, because the owner node will keep track of the list of node holding a valid copy of the region. This greatly reduce network traffic, because the network is a point-to-point type and not a bus. Another feature is that the *owner coherence metadata* moves with the owner node. Most DSM systems define an *home node* at region creation, which does not change, and stores owner metadata for the region. This can lead to performance problem if the region is mostly used by other nodes: they will have to contact the home node frequently to maintain the metadata up-to-date. In Givy, the creation node is called a *creator*: it only tracks the current owner node location, and in many case is not required to locate the owner node.

One important property is that any node should be able to contact the owner node at anytime. This is achieved using three mechanisms:

- The initial owner (*creator*) is always the node which allocated the region. The allocator subsystem of Givy provides a *constant time node local* way to determine the creator node from its address (Section 4.2).

- When ownership changes, the old owner node knows which node is the new owner, and will set its internal state to believe that the new node will be the owner. By doing so, if we start at the creator node (initial owner), and follow iteratively nodes guess of which node is the owner, we will always end up finding the current owner.

- Protocol messages whose destination is the owner can then be forwarded using this chain, and will reach the owner. If a node without any idea of who the owner is tries to emit such a message, it can always safely send it to the creator node. The protocol also reduce the chain length by informing the creator of ownership changes as an optimization.

Snooping MSI requires atomicity of bus events. Givy operates on a network which can have delays, and we only assume messages will eventually arrive (no packet lost or duplicated, but reordering allowed). So we must ensure that a node with a task accessing a region in RW mode will invalidate the copies of that region on other nodes before tasks which depend on the current task can start, and eventually read the new data in the region. This is done by adding a memory semantic to the `task_tdec` operation: tasks dependencies will be released only when modifications have been published globally (invalidations).

3.3.1 Formal Model

This section describes the formal protocol. It first shows how the global state is structured, then describes rules in three groups: network, cache, tasks. The formalization is

written using a nice tool called OTT [65], which generated the LaTeX macros to show transition rules.

Structures

The global state is contained in the **s** variable. This state variable is an array of node states, accessed by $\mathbf{s}[n]$ with n a node identifier. Note that contrary to previous protocols which abstracted the processor (and program structure) entirely, this one must explicit the task dependencies as they are used to control protocol messages.

The state of a node contains:

- **rq**: the network interface receive queue.
- **sq**: the network interface send queue.
- **r**: a map from region addresses to region descriptors (which store the protocol data on the region).
- **tasks**: the set of tasks (descriptors) currently owned by the node.
- **tdec**: map from pair of tasks to a set of addresses. It is filled by the current task with addresses that a `task_tdec` from this task to the second task should wait to be published.

Each region descriptor contains:

- **own**: a node identifier that should point (transitively) on the current owner. It can be the local node, meaning we are owning this region, and defaults to the allocator of that address.
- **valid**: boolean telling if the stored data (abstracted by **d**) is valid (the owner should have a true flag).
- **vs**: *valid set*, this is the directory part of the protocol (owner metadata), which holds the list of nodes with a valid copy. It should only be non empty for the owner, and is only used by it. The owner should never be inside this set.
- **wias**: *waiting invalidation acknowledge set*: set of flags indicating that an invalidation is in flight (owner metadata).
- **req**: *requested flag*: used to indicate that a request has been sent (for many requests).

A task descriptor has two fields. The first is **r**, a map from region to access mode which describes which addresses are accessed and in which mode. The second indicates in what step of execution the task is. These steps will be described along with the corresponding transitions in the next section.

Tasks are not globally visible in the runtime: they are owned by one node at a time, but can move at some steps of their execution. Moving tasks from one node to another is handled by the scheduling part of the run-time, which will be abstracted as it is not our focus here. Thus in this formalization task descriptor coherence is not addressed. All the transitions are atomic.

Network

This first rule is used to abstract the network. It just removes a message from a send queue of a node, and add it atomically to the receive queue of the destination node:

$$\frac{\textbf{dequeue}\left(\textbf{s}[n].\,\textbf{sq}\right) = (\textit{to}, \textit{msg})}{\textbf{enqueue}\left(\textbf{s}[\textit{to}].\,\textbf{rq}, \textit{msg}\right)} \quad \text{ASYNC_SEND}$$

Reactions to Network Events

The **DataReq** and **DataAns** messages are equivalent to the BusRd event in MSI. The first is sent to request fresh data, and the second is used by the owner to send the fresh data back to the requester.

The first of these three rules forwards the request to the owner in case the node n is not the owner. The second rule is used when the owner receives the request. In this case, we answer with the data, and then we add the node who requested it to the list of valid node (it will get a valid copy once the answers arrives). The last rule updates the requester region descriptor according to the answer. It stores the new data, sets the valid flag as true, and resets the *requested* flag. As only the owner could send this message, we update our owner pointer to shorten the path to it (optimization).

$$\frac{\begin{array}{l}\textbf{dequeue}\left(\textbf{s}[n].\,\textbf{rq}\right) = \textbf{DataReq}\left(\textit{from}, \textit{addr}\right)\\ \textbf{s}[n].\,\textbf{r}\left[\textit{addr}\right] = \textit{reg} \wedge \textit{reg}.\,\textbf{own} \neq n\end{array}}{\textbf{enqueue}\left(\textbf{s}[n].\,\textbf{sq}, \left(\textit{reg}.\,\textbf{own}, \textbf{DataReq}\left(\textit{from}, \textit{addr}\right)\right)\right)} \quad \text{FORWARD_DATA_REQ}$$

$$\frac{\begin{array}{l}\textbf{dequeue}\left(\textbf{s}[n].\,\textbf{rq}\right) = \textbf{DataReq}\left(\textit{from}, \textit{addr}\right)\\ \textbf{s}[n].\,\textbf{r}\left[\textit{addr}\right] = \textit{reg} \wedge \textit{reg}.\,\textbf{own} = n\end{array}}{\begin{array}{l}\textit{reg}.\,\textbf{vs} := \textit{reg}.\,\textbf{vs} \cup \{\textit{from}\}\\ \textbf{enqueue}\left(\textbf{s}[n].\,\textbf{sq}, \left(\textit{from}, \textbf{DataAns}\left(n, \textit{addr}, \textit{reg}.\,\textbf{d}\right)\right)\right)\end{array}} \quad \text{HANDLE_DATA_REQ}$$

$$\frac{\textbf{dequeue}\left(\textbf{s}[n].\,\textbf{rq}\right) = \textbf{DataAns}\left(\textit{from}, \textit{addr}, \textit{data}\right)}{\begin{array}{l}\textbf{s}[n].\,\textbf{r}\left[\textit{addr}\right] := \{\\ \quad \textbf{valid} : \textbf{true}, \textbf{own} : \textit{from}, \textbf{vs} : \emptyset,\\ \quad \textbf{wias} : \emptyset, \textbf{req} : \textbf{false}, \textbf{d} : \textit{data}\\ \}\end{array}} \quad \text{HANDLE_DATA_ANS}$$

The next three rules handle ownership requests and transfers, and are similar to the BusM event in MSI. In snooping MSI the bus guaranteed that only one busM event could be used. Here, the properties of the `task_tdec` operations ensures that no tasks can access a region that is being accessed in RW mode (there would be a chain of dependencies between the two, so they cannot be executed at the same time). So we are sure that no ownership change can occur if the data is being used elsewhere, and that no one uses the data when ownership change occurs.

The first one is a forwarding rule, like the one above. The second one handles ownership giveaway. The node will give its ownership of the region by setting the requester node as the owner in the owner pointer, maintaining the chain to the owner. The old owner

node will go in a valid copy state by setting valid flag to true, and give the ownership to the requester by sending a message.

The third rule is used when the requester receives the ownership transfer message. It then set itself as the owner (valid flag to true, fresh data sent by the previous owner, and owner pointer to itself). Another meta-data transmitted is the directory information, the valid set. It is updated to take into account that the previous owner is now sharing a valid copy.

There is a period of time where no node is declared as owner, which can pose problems if requests to owner are sent. Such requests can be DataReq/DataAns, OwnReq/OwnTrans, or InvalidReq/InvalidAns (see below). But there cannot be any of them as this would mean that another task is accessing the same region: this task would be ordered by dependencies and would not be executable.

$$\dfrac{\begin{array}{l}\mathbf{dequeue}\,(\mathbf{s}[n].\,\mathbf{rq}) = \mathbf{OwnReq}\,(\mathit{from}, \mathit{addr})\\ \mathbf{s}[n].\,\mathbf{r}\,[\mathit{addr}] = \mathit{reg} \wedge \mathit{reg}.\,\mathbf{own} \neq n\end{array}}{\mathbf{enqueue}\,(\mathbf{s}[n].\,\mathbf{sq}, (\mathit{reg}.\,\mathbf{own}, \mathbf{OwnReq}\,(\mathit{from}, \mathit{addr})))}\ \text{FORWARD_OWN_REQ}$$

$$\dfrac{\begin{array}{l}\mathbf{dequeue}\,(\mathbf{s}[n].\,\mathbf{rq}) = \mathbf{OwnReq}\,(\mathit{from}, \mathit{addr})\\ \mathbf{s}[n].\,\mathbf{r}\,[\mathit{addr}] = \mathit{reg} \wedge \mathit{reg}.\,\mathbf{own} = n\end{array}}{\begin{array}{l}\mathbf{s}[n].\,\mathbf{r}\,[\mathit{addr}] := \{\\ \quad \mathbf{valid} : \mathbf{true}, \mathbf{own} : \mathit{from}, \mathbf{vs} : \emptyset,\\ \quad \mathbf{wias} : \emptyset, \mathbf{req} : \mathbf{false}, \mathbf{d} : \mathit{reg}.\,\mathbf{d}\\ \}\\ \mathbf{enqueue}\,(\mathbf{s}[n].\,\mathbf{sq}, (\mathit{from},\\ \quad \mathbf{OwnTrans}\,(n, \mathit{addr}, \mathit{reg}.\,\mathbf{vs} \cup \{n\}, \mathit{reg}.\,\mathbf{d})))\end{array}}\ \text{HANDLE_OWNER_REQ}$$

$$\dfrac{\mathbf{dequeue}\,(\mathbf{s}[n].\,\mathbf{rq}) = \mathbf{OwnTrans}\,(\mathit{from}, \mathit{addr}, \mathit{set}, \mathit{data})}{\begin{array}{l}\mathbf{s}[n].\,\mathbf{r}\,[\mathit{addr}] := \{\\ \quad \mathbf{valid} : \mathbf{true}, \mathbf{own} : n, \mathbf{vs} : \mathit{set} \setminus \{n\},\\ \quad \mathbf{wias} : \emptyset, \mathbf{req} : \mathbf{false}, \mathbf{d} : \mathit{data}\\ \}\end{array}}\ \text{HANDLE_OWNER_TRANSFER}$$

Invalidations complement the ownership transfer process. In MSI, the busM event handles both M state position and invalidation of others caches. Here, we must explicitly send invalidate requests to others caches upon modification, as we use a directory protocol.

The first transition invalidates the foreign cache, which acknowledge the sender, and takes this opportunity to shorten the owner path again. The second transition is when we receive acknowledgement. Invalidations are sent in a step where the owner will invalidate all nodes in the valid set, and use the **wias** field to list nodes which have not yet acknowledged (as its name, *waiting invalidation acknowledge set* suggested). So we remove the node from this set.

$$\dfrac{\begin{array}{l}\mathbf{dequeue}\,(\mathbf{s}[n].\,\mathbf{rq}) = \mathbf{InvalidReq}\,(\mathit{from}, \mathit{addr})\\ \mathbf{s}[n].\,\mathbf{r}\,[\mathit{addr}] = \mathit{reg}\end{array}}{\begin{array}{l}\mathit{reg}.\,\mathbf{valid} := \mathbf{false}\\ \mathit{reg}.\,\mathbf{own} := \mathit{from}\\ \mathbf{enqueue}\,(\mathbf{s}[n].\,\mathbf{sq}, (\mathit{from}, \mathbf{InvalidAck}\,(n, \mathit{addr})))\end{array}}\ \text{HANDLE_INV_REQ}$$

$$\frac{\mathbf{dequeue}\,(\mathbf{s}[n].\,\mathbf{rq}) = \mathbf{InvalidAck}\,(\mathit{from}, \mathit{addr})}{\mathbf{s}[n].\,\mathbf{r}\,[\mathit{addr}].\,\mathbf{wias} := \mathbf{s}[n].\,\mathbf{r}\,[\mathit{addr}].\,\mathbf{wias} \setminus \{\mathit{from}\}} \quad \text{HANDLE_INV_ACK}$$

Execution Model Transitions

This is the last part of the model, which covers how tasks generate protocol events. A newly created task is owned by the node where it has been created. It starts in the in the **Created** state (in *task*.**s**): it is waiting its dependencies to be fulfilled. Dependencies will be fulfilled later by parent tasks, by using `task_tdec` operations. When all needed `task_tdec` operations are performed, the task becomes eligible:

$$\frac{\begin{array}{l} \mathit{task} \in \mathbf{s}[n].\,\mathbf{tasks} \wedge \mathit{task}.\,\mathbf{s} = \mathbf{Created} \\ \mathbf{allTDecDone}\,(\mathit{task}) \end{array}}{\mathit{task}.\,\mathbf{s} = \mathbf{Eligible}} \quad \text{TASK_BECOMES_ELIGIBLE}$$

In the **Eligible** state, a task has no more dependencies left and can be executed. First a node must be select for execution: this is a scheduling step, which is modeled by the first rule below (task can be moved from node to node). After a node is selected, the *caching* phase starts.

$$\frac{\mathit{task} \in \mathbf{s}[n].\,\mathbf{tasks} \wedge \mathit{task}.\,\mathbf{s} = \mathbf{Eligible}}{\begin{array}{l} \mathbf{s}[n].\,\mathbf{tasks} := \mathbf{s}[n].\,\mathbf{tasks} \setminus \{\mathit{task}\} \\ \mathbf{s}[\mathit{aNodeId}].\,\mathbf{tasks} := \mathbf{s}[\mathit{aNodeId}].\,\mathbf{tasks} \cup \{\mathit{task}\} \end{array}} \quad \text{TASK_MOVED}$$

$$\frac{\mathit{task} \in \mathbf{s}[n].\,\mathbf{tasks} \wedge \mathit{task}.\,\mathbf{s} = \mathbf{Eligible}}{\mathit{task}.\,\mathbf{s} = \mathbf{Caching}} \quad \text{TASK_STARTS_CACHING}$$

Requests are sent for regions not already available as valid copies on the node. Task execution will only start when all regions are locally available in the requested access modes. **DataReq** messages are sent for read only accessed regions. **OwnReq** messages are sent for write accessed regions which are not owned. Starting from the caching step, the task is fixed to its execution node and will not move anymore (it would be inefficient to move while data is being loaded locally). The *requested* flag is used to send only one request per region.

$$\frac{\begin{array}{l} \mathit{task} \in \mathbf{s}[n].\,\mathbf{tasks} \wedge \mathit{task}.\,\mathbf{s} = \mathbf{Caching} \wedge \mathit{task}.\,\mathbf{r}\,[\mathit{addr}] = \mathbf{R} \\ \mathbf{s}[n].\,\mathbf{r}\,[\mathit{addr}] = \mathit{reg} \wedge \mathbf{not}\ \mathit{reg}.\,\mathbf{valid} \wedge \mathbf{not}\ \mathit{reg}.\,\mathbf{req} \end{array}}{\begin{array}{l} \mathit{reg}.\,\mathbf{req} := \mathbf{true} \\ \mathbf{enqueue}\,(\mathbf{s}[n].\,\mathbf{sq}, (\mathit{reg}.\,\mathbf{own}, \mathbf{DataReq}\,(n, \mathit{addr}))) \end{array}} \quad \text{TRIGGER_DATA_REQ}$$

$$\frac{\begin{array}{l} \mathit{task} \in \mathbf{s}[n].\,\mathbf{tasks} \wedge \mathit{task}.\,\mathbf{s} = \mathbf{Caching} \wedge \mathit{task}.\,\mathbf{r}\,[\mathit{addr}] = \mathbf{RW} \\ \mathbf{s}[n].\,\mathbf{r}\,[\mathit{addr}] = \mathit{reg} \wedge \mathit{reg}.\,\mathbf{own} \neq n \wedge \mathbf{not}\ \mathit{reg}.\,\mathbf{req} \end{array}}{\begin{array}{l} \mathit{reg}.\,\mathbf{req} := \mathbf{true} \\ \mathbf{enqueue}\,(\mathbf{s}[n].\,\mathbf{sq}, (\mathit{reg}.\,\mathbf{own}, \mathbf{OwnReq}\,(n, \mathit{addr}))) \end{array}} \quad \text{TRIGGER_OWN_REQ}$$

When all requests have been completed and all regions locally available in their respective modes, the task can be executed (**Running** state). There is no limit on how much tasks can be running simultaneously in this model. In the runtime, the task would be put into a node *ready queue*, and then one of the *worker threads* would execute it. However this has no impact on the coherence protocol, so it is left out of the protocol itself. The absence of data race ensures no other task will invalidate or take ownership of any accessed region before we can execute the task.

$$\frac{\begin{array}{l} \textit{task} \in \mathbf{s}[n].\,\mathbf{tasks} \wedge \textit{task}.\,\mathbf{s} = \mathbf{Caching} \\ \mathbf{forall}\,(\textit{addr})\{\textit{task}.\,\mathbf{r}\,[\textit{addr}] = \mathbf{R} \Rightarrow \mathbf{s}[n].\,\mathbf{r}\,[\textit{addr}].\,\mathbf{valid}\} \\ \mathbf{forall}\,(\textit{addr})\{\textit{task}.\,\mathbf{r}\,[\textit{addr}] = \mathbf{RW} \Rightarrow \mathbf{s}[n].\,\mathbf{r}\,[\textit{addr}].\,\mathbf{own} = n\} \end{array}}{\textit{task}.\,\mathbf{s} := \mathbf{Running}} \;\text{TASK_STARTS_RUNNING}$$

During execution, the task code will register `task_tdec` operations in the **tdec** field of the node, with the addresses of regions that must be *published*. After execution, the task will enter the **Publishing** state, where it will invalidate all copies of modified regions, and then complete the `task_tdec` operations.

$$\frac{\textit{task} \in \mathbf{s}[n].\,\mathbf{tasks} \wedge \textit{task}.\,\mathbf{s} = \mathbf{Running}}{\textit{task}.\,\mathbf{s} := \mathbf{Publishing}} \;\text{TASK_STARTS_PUBLISHING}$$

Invalidations requests are sent. The **wias** set indicates the set of non-completed invalidations.

$$\frac{\begin{array}{l} \textit{task} \in \mathbf{s}[n].\,\mathbf{tasks} \wedge \textit{task}.\,\mathbf{s} = \mathbf{Publishing} \wedge \textit{task}.\,\mathbf{r}\,[\textit{addr}] = \mathbf{RW} \\ \mathbf{s}[n].\,\mathbf{r}\,[\textit{addr}] = \textit{reg} \wedge t \in \textit{reg}.\,\mathbf{vs} \end{array}}{\begin{array}{l} \textit{reg}.\,\mathbf{vs} := \textit{reg}.\,\mathbf{vs} \setminus \{t\} \\ \textit{reg}.\,\mathbf{wias} := \textit{reg}.\,\mathbf{wias} \cup \{t\} \\ \mathbf{enqueue}\,(\mathbf{s}[n].\,\mathbf{sq}, (t, \mathbf{InvalidReq}\,(n, \textit{addr}))) \end{array}} \;\text{TRIGGER_INV_REQ}$$

When all invalidations for a region have been completed, we can perform the deferred `task_tdec` operations:

$$\frac{\begin{array}{l} \textit{task} \in \mathbf{s}[n].\,\mathbf{tasks} \wedge \textit{task}.\,\mathbf{s} = \mathbf{Publishing} \\ \textit{addr} \in \mathbf{s}[n].\,\mathbf{tdec}\,[\textit{task} \to \textit{task}'] \\ \mathbf{s}[n].\,\mathbf{r}\,[\textit{addr}] = \textit{reg} \wedge \textit{reg}.\,\mathbf{vs} = \emptyset \wedge \textit{reg}.\,\mathbf{wias} = \emptyset \end{array}}{\mathbf{s}[n].\,\mathbf{tdec}\,[\textit{task} \to \textit{task}'] := \mathbf{s}[n].\,\mathbf{tdec}\,[\textit{task} \to \textit{task}'] \setminus \{\textit{addr}\}} \;\text{ALL_INV_ACK_RECEIVED}$$

To simplify the model syntax, we assume that registering a `task_tdec` in the **tdec** table adds at least a keyword **dep** in the set of dependencies. If there is only this keyword left, this means that all invalidations for this `task_tdec` where solved, and then we can perform it. In the model, performing a `task_tdec` is locally done using the **doTDec**

primitive. So if the task is not owned by this node, we must use a message to have the owner node do the `task_tdec`:

$$\frac{\begin{array}{l} task \in \mathbf{s}[n].\,\mathbf{tasks} \land task.\,\mathbf{s} = \mathbf{Publishing} \\ task' \in \mathbf{s}[n].\,\mathbf{tasks} \\ \mathbf{s}[n].\,\mathbf{tdec}\,[task \rightarrow task'] = \{\mathbf{dep}\} \end{array}}{\begin{array}{l} \mathbf{s}[n].\,\mathbf{tdec}\,[task \rightarrow task'] := \emptyset \\ \mathbf{doTDec}\,(task') \end{array}} \quad \text{TRIGGER_LOCAL_TDEC}$$

$$\frac{\begin{array}{l} task \in \mathbf{s}[n].\,\mathbf{tasks} \land task.\,\mathbf{s} = \mathbf{Publishing} \\ task' \notin \mathbf{s}[n].\,\mathbf{tasks} \\ \mathbf{s}[n].\,\mathbf{tdec}\,[task \rightarrow task'] = \{\mathbf{dep}\} \end{array}}{\begin{array}{l} \mathbf{s}[n].\,\mathbf{tdec}\,[task \rightarrow task'] := \emptyset \\ \mathbf{enqueue}\,(\mathbf{s}[n].\,\mathbf{sq}, (\mathbf{creator}\,(task'), \mathbf{PerformTDec}\,(task'))) \end{array}} \quad \text{TRIGGER_REMOTE_TDEC}$$

$$\frac{\mathbf{dequeue}\,(\mathbf{s}[n].\,\mathbf{rq}) = \mathbf{PerformTDec}\,(task)}{\mathbf{doTDec}\,(task)} \quad \text{HANDLE_PERFORM_TDEC}$$

When all `task_tdec` have been performed, the task has been completed, and we can delete it:

$$\frac{\begin{array}{c} task \in \mathbf{s}[n].\,\mathbf{tasks} \land task.\,\mathbf{s} = \mathbf{Publishing} \\ \mathbf{forall}\,(task')\{\mathbf{s}[n].\,\mathbf{tdec}\,[task \rightarrow task'] = \emptyset\} \end{array}}{task.\,\mathbf{s} := \mathbf{Finished}} \quad \text{TASK_FINISHED}$$

3.3.2 Discussion and Extensions

The current protocol has been created from a prototype OWM implementation from Boris Arnoux [17]. The prototype tried to provide an access mode with multiple writers on exclusive parts of the same region. Such a mode greatly increased protocol complexity and performance overhead (more messages on the critical path to task completion for example). In Givy, this has been replaced by the splitting extension described in Section 2.4.1. In this extension regions can be split into independent parts with respect to coherence properties: it allows concurrent exclusive writers in a safer and less expensive context, without breaking the exclusive ownership design. Splitting a region requires ownership, and semantically destroys the region to create a set of new regions (the splitted parts). A split will invalidate copies due to normal ownership mechanisms, and subsequent access request answers will have a special tag indicating requester nodes to consider the region as splitted (and update their local metadata). Merging a splitted region requires ownership of all parts, which are destroyed and replaced by the initial region.

`task_set_arg` may also perform concurrent exclusive writes, if multiple tasks are filling different arguments of a new task. Task frames (storing arguments and metadata) are implemented as special regions in the current model. In this case splitting is not very effective, and a special mode reserved to `task_set_arg` could be introduced. Such a mode would be implemented by directly sending the argument to the task frame owner node by a TaskSetArg message. An even better optimization is to pack a TaskSetArg

message content with the message used to perform a `task_tdec` that would normally follow: they have the same target, which is the task frame.

The memory allocator of Givy is responsible for managing the memory placement of the coherence protocol metadata. This placement is described in Section 4.5.3 for the version two of the allocator subsystem. Due to the design of the allocator, additional allocator metadata (layout, sizing) must be transmitted by the coherence protocol. It only requires slightly bigger coherence message when data is transmitted, and does not change anything else in the protocol.

RDMA (Section 2.3.3) has not been addressed by the formalization. RDMA transfers would be used only for big region transfers that do not fit small messages. The formal DataAns message would become a multiple phase operation, with an RDMA initiation, the actual data transfer, and a finish phase to update metadata. RDMA transfers have not been implemented in the unfinished Givy runtime.

An interesting extension is to include the host node in the GAS for accelerator-like systems (MPPA). It could be tagged as having a lot of memory space, and thus could be used as a default storage unit when memory must be freed on the small accelerator system. This would require extensions to the protocol to be able to push unused regions toward this special node when space is scarce.

Attempts to validate that the current protocol is valid are described in Chapter 5. The choice of an operational model was partly made to potentially enable coherence protocol code generation from the formalization. Such code generation would be very practical for testing multiple designs without expensive and time consuming rewrites of complex code. However the generated code would have to integrate properly with the network subsystem, and the allocator subsystem which were not yet developed at the time. Thus code generation was not done for the initial prototype Givy runtime, and is left for future work.

Chapter 4

Memory Allocator

This chapter discusses the memory allocation subsystem of Givy. Memory allocator design is important for two aspects: raw dynamic memory allocation performance, and providing support for the global address space of Givy. Section 4.1 covers classic aspects of memory allocation. Section 4.2 discusses impact of the global address space on the memory allocator. Section 4.3 describes current memory allocators, while Section 4.4 and Section 4.5 describes the two versions of the Givy allocator subsystem.

4.1 Memory Allocation

Memory allocation is the art of choosing the layout of program objects in memory. Memory can be viewed as an array (big or small) of cells, which can be indexed by an integer value : an *address*. Thus memory allocation is, given a set of required program objects (each with a size), finding an address for each one such that there are no overlaps between objects memory segments. In our runtime system, we must address the problem of memory allocation in the global address space, which is a linear interval of addresses.

Memory allocation can be *static* or *dynamic*. Static memory allocation is allocation done at compile time, and requires compile time size information on objects. It is very specific, and happens for example when a C compiler must organize the stack variables in a function. Other uses include static allocation of buffers in very optimized code, like code encountered after polyhedral tiling, or stream application compilation. However, most application have memory requirement – size and number of objects – that depend on program input. To handle these programs, we must place objects at runtime, using *dynamic allocation*. In the following chapter we will only cover dynamic allocation techniques.

In the following chapter, we will target a POSIX machine with virtual memory support (see Section 2.1). In this context, *memory allocator* refers to the user-space part of the memory management stack. Such a memory allocator has two main roles. The first role is to manage the virtual memory state of the heap, using the OS system calls `mmap()`/`munmap()` or `brk()` . The second role is to internally divide the pieces of virtual memory to fulfill individual allocation requests. As system calls are costly, good allocators call them as little as possible. Thus they usually create/destroy virtual memory in big chunks, and manage these chunks as a *cache* of unused memory.

The next section describes the POSIX C API of memory allocators. Then performance

considerations will be discussed. Finally, we describe some basic blocs used to build most allocators today.

4.1.1 API

In the C language, dynamic memory allocation must be handled by the programmer. It is provided by the *standard library* (libc), and is used through the API in Listing 4.1. It uses manual function calls that incrementally create and destroy individual blocks. The following API description is not exhaustive, and omits older mostly deprecated alternatives to `posix_memalign()`, and functions like `calloc()` that can be trivially built on top of the described API.

Listing 4.1: Dynamic memory allocation API in C

```c
// POSIX, basic
void* malloc(size_t size);
void free(void* addr);
// POSIX, advanced
void* realloc(void* addr, size_t size);
int posix_memalign(void** addr_ptr, size_t alignment, size_t size);
// Non POSIX
size_t malloc_usable_size(void* addr); // from libc
void free_sized(void* addr, size_t size); // from me
```

`malloc(size)` *creates* a new block measuring `size`, and returns its address. In other words, it reserves the memory segment $[addr, addr + size[$ for the caller of `malloc()`, and ensures that the memory is usable. `free(addr)` takes the address of an allocated block, and *destroys* it. It means releasing it and making the memory segment $[addr, addr + size[$ available for future allocations.

`malloc_usable_size(addr)` returns the size of the block that has been reserved, which may exceed the requested size depending on the allocator implementation. Old allocator implementation tended to give perfect-fit blocks, but new implementations will usually return more for performance reasons (see 4.1.3). As `free()` does not have any `size` argument, implementations must be able to deduce the size of a block from their address, usually by reading some metadata. `free_sized(addr, size)` is identical to `free()`, but provides the size to the implementation, allowing it to potentially skip some metadata reads.

`calloc(addr, size)` "resizes" the block at `addr` to the new size. It returns the address of a block that has the requested size, the data of the given block (truncated to the minimum of the old and new size), and then destroys the old block. It used mostly for buffer resizing, and may return the same block if it fits the new size. Thus function call is not required – it can be implemented in terms of `malloc()`, `memcpy()` and `free()`– but it lets allocator implementation optimize this specific operation if their structure allows it.

`posix_memalign(&addr, size, align)` creates a block like `malloc()`, but requires that the block address `addr` is *aligned* on `align` bytes (which must be a power of 2). A

memory address `addr` is said to have an alignment of `align` if and only if `addr % align == 0`. Alignment is used to represent processor or memory system constraints [35]. Some processor instructions that access memory may require specific alignment on memory addresses, or just be more efficient / faster if they are used on properly aligned addresses. In C, every data type has an alignment value, computed by the compiler (it can be queried by the `alignof(T)` macro). Every piece of data should be aligned to this value at runtime. Stack values will be automatically aligned by the compiler. However heap values alignments are determined by the allocator implementation. Usually `malloc()` will align its blocks to a "good enough" alignment (like 8 bytes on x86_64 architectures), and `posix_memalign()` is used to force higher alignment requirements (16 bytes for SSE simd instructions, or double word CAS, ...). A naive implementation of `posix_memalign()` can be seen in Listing 4.2, but wastes memory and requires that `free()` handles in-block pointers correctly. Allocator implementations can, again, provide an optimized version.

Listing 4.2: Naive implementation of `posix_memalign()`

```
int posix_memalign(void** addr, size_t size, size_t align) {
  void* block = malloc(size + align - 1);
  // block contains a size sized interval that is correctly aligned
  *addr = align_up(block, align); // find it and return it
  return 0;
}
void* align_up(void* addr, size_t align) {
  // find the next aligned pointer >= addr
  uintptr_t aligned_ptr_mask = ~(align - 1);
  return (void*) ((align - 1 + (uintptr_t) addr) & aligned_ptr_mask);
}
```

Misalignment of data can cause surprising bugs: During the development of the V1 of the Givy allocator, I forgot to force a 16 bytes alignment on some internal structures. It later caused a *segmentation fault* at a perfectly valid and accessible memory address. The culprit was a double word CAS (compare-and-swap) instruction, generated from C11 atomics, accessing these structures. It required 16 bytes aligned addresses and failed due to misalignment.

To work, Givy needs dynamic allocation of global objects (`malloc()`/ `free()`), and task access annotations. Thus it could be extended to other programming languages as long as they rely on the `malloc()` and `free()` interface for dynamic objects, and provide task annotations. For example C++ `new` and `delete` operators use `malloc()` and `free()` but also call object constructors and destructors. When Givy moves data between node, it moves the physical location of the object memory, but does not perform a logical copy or move of the object itself. Thus Givy does not need to be aware of C++ objects copy / move constructors. So Givy could support C++ objects as regions, as long as they have been dynamically allocated to the heap. This could be extended to other languages, as long as they have a notion of contiguous memory representation of objects, and dynamically allocate global objects using `malloc()`.

4.1.2 Performance Criteria And Trade-offs

We will now discuss the performance constraint of an allocator implementation, and trade-offs between these constraints.

Allocation Speed

The first performance criteria is the time taken by `malloc()` or `free()` to process the request. This time is determined by the allocator data structures (metadata), the caching strategy and the type of multi-threading synchronization.

Allocator data structures are accessed during each call to `malloc()` (to find a new block in unused virtual memory) or `free()` (to find where to place the freed block in structures). Their access time should have a small complexity, constant time if possible. They should also be designed to be cache-friendly (by avoiding too much pointer indirections, for example).

Allocators use `mmap()`/ `munmap()` to create or destroy virtual memory. These systems calls are slow, so implementations should call them as rarely as possible. A usual strategy is to create or destroy virtual memory in big chunks compared to the size of allocations. If these chunks are N times bigger than the average allocation size, then `malloc()` will call the slow `mmap()` only every N requests. The bigger N is, the faster `malloc()` is (in amortized time), but at the cost of a bigger average memory usage. The same applies to `free()` and `munmap()`.

And finally, in a multi-threaded allocator, the choice of the synchronization type will impact allocation speed. Just wrapping a non multi-threaded `malloc()` with a mutex works, but slow down very fast under contention. No synchronization – equivalent to one allocator instance per thread – suffers less from contention, but uses more memory as allocators cannot share their unused virtual memory chunks. Current allocator structures are in the middle, with some thread-private structures and some shared structures. They also tend to use *lock-free* structures that offer high scalability and suffer less from contention. Some can even do away with synchronizations in some cases, which is good as synchronization primitives are will frequently flush the hardware caches.

Application Speed Due to Layout Quality

The overall performance of a program depends on the memory layout generated by the memory allocator.

The memory layout should be as compact as possible to help the hardware cache and the TLB (see Section 2.1). If frequently accessed memory blocks are spread over more pages than the TLB can fit, it will cause a lot of page faults which slows the program. The same effect can happen at cache level. Allocator metadata is only accessed during `malloc()` and `free()` operations, and useless during the rest of program execution. Modern allocators like SFMalloc [64] tend to segregate metadata and program data. However, older allocators like dlmalloc [42] mixed data and metadata more closely (each block was surrounded on both side by some metadata). So program data is spread over more cache lines by memory which will not be accessed during computation, and risk triggering more cache misses.

Another concept impacting program performance is memory *reuse*. When requesting a memory block with `malloc()`, it is a good idea to give a block of memory which was freed recently. This block is likely to be in the cache and TLB compared to a fresh block, and accesses will be faster. Thus good allocators should be able to recycle recently freed memory as soon as possible.

Finally, multi-threaded allocators should avoid *false sharing*. False sharing occurs when two processors use different parts of the same cache line. Due to hardware cache coherence design, these accesses will slow down a lot. To completely remove false sharing, allocators must pad every block to take entire cache lines, which is wasteful. Most allocators will try to prevent creating false sharing situations, by giving all parts of a cache line to the same thread. A discussion of false-sharing avoidance techniques is in the Hoard paper [21].

Memory Efficiency

Memory efficiency of an allocator can be defined as the ratio between peak program requested memory and peak virtual memory allocated from the system for a given program execution. Conversely, the Hoard paper [21] defines *fragmentation* as the opposite ratio. A better memory efficiency means that program can process bigger data sets on a given machine, and may be faster by avoiding TLB or cache misses. It is impacted mostly by the caching strategy, the layout, and the metadata overhead.

Metadata takes space but is not used by the program itself, so an allocator should try to use a small metadata amount per allocation / byte of allocation. Old implementations like dlmalloc [42] had a fixed amount of metadata per allocation, which is very costly for a big number of small allocations.

The caching strategy impacts memory usage a lot. Reducing memory fragmentation means increasing the opportunities of reuse of blocks in the allocator. It can be reuse between thread private heap structures, which will cost more in synchronizations. It can be reuse between different caching structures, which means more structures to check at each `malloc()` or `free()`. In both cases, there is a trade-off between memory efficiency and speed.

And obviously, memory layout itself impacts memory efficiency. During cycles of `malloc()` and `free()`, the virtual address space can become fragmented, which will prevent reusing blocks if they are all smaller than the request. In addition to that, most allocators nowadays use the concept of slab allocation, which adds a small efficiency cost but simplifies structures and improves speed.

4.1.3 Classic Allocator Structures

Free List

A very basic by useful and efficient structure is the *free list*. A free list is a single linked list of unused memory blocks. The first `sizeof(void*)` bytes of each block are used as a `next` pointer in a classic single linked list. The blocks are then threaded in a list, that only needs a single external pointer to reference the head. The blocks memory can be used by the allocator because they are not used by the program.

There are many variants which add element counting, additional information per unused block (like its size), double linked list, tree structure, lock-free list... Every block must contain at least `sizeof(void*)` bytes – more if the structure is more complicated – which puts a lower bound on the smallest allocation size.

Slab Allocator

A *slab allocator* is a very simple allocator that only allocates a specific size of block. It contains a free list to store freed blocks. `malloc()` will try to get a block from the free list. If it fails it gets a chunk of virtual memory from `mmap()`, cuts it into blocks of the slab size, adds the blocks to the free lists and give one of them to the requester. `free(ptr)` will add `ptr` to the free list.

Advanced variants can cut the blocks progressively to prevent touching too many cache lines at once. They can also use one counted free list per virtual memory chunk to detect totally unused chunks and `munmap()` them. The size of virtual memory chunks is a trade-off between speed and memory usage.

Segregated Slab Allocator

A *segregated slab allocator* is composed of a set of slab allocators. These slab allocators are configured according to a size scale. `malloc(s)` will compare `s` against the scale and use the smallest fitting sized slab allocator to answer the request. `free(ptr)` has to discover the block size of `ptr` before freeing with the right slab allocator. The size can be detected by comparing the `ptr` to a table of virtual memory chunks, or by using a size metadata near the block, or given directly by `free_sized()`.

Segregated slab allocators forms the basis of most modern high performance allocators, at least for small sizes allocations. They are optimized for speed, with very simple structures and few metadata. Some memory is wasted if requests are between two sizes classes (*internal fragmentation*). This can be reduced if size classes are numerous and with small size intervals. However in this case there is a risk of creating a lot of virtual memory chunks with low memory usage in many size classes if requests are random.

Thread Heap & Arenas

Multi-threaded allocator heap structures must be designed to reduce contention as much as possible. A very common design choice is to split the front-end heap structure into multiple instances of this heap structure. Then reducing the contention can be done by spreading requests among the pool of heap structures.

If each thread has its own heap structure, this is called *thread heaps* or *private heaps*. Thread heaps require little to no synchronization (synchronization is pushed to the next centralized heap structure). They can be accessed efficiently using *thread local storage* variables (supported in modern C/C++/pthreads). However the allocator must manage thread creation and death to avoid leaking memory.

If heap structures may be shared by some threads, they are called *Arenas*. This design keeps some synchronization at the front-end heap structures, but still spread contention among arenas. The arena number is usually chosen as a function of the number of cores.

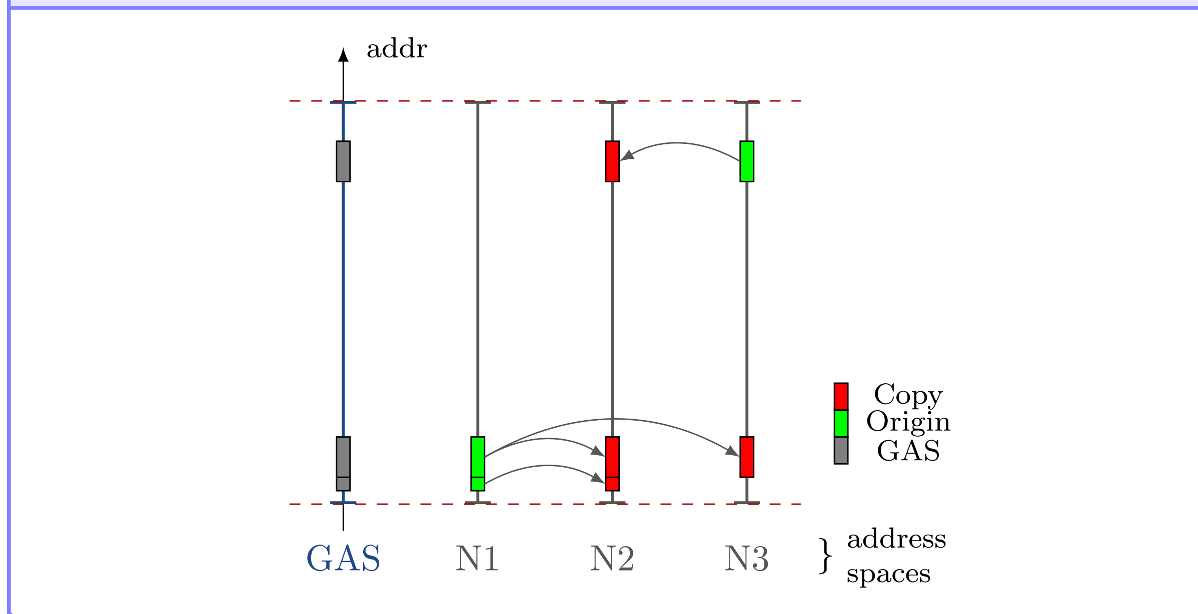
4.2 Global Address Space Management

GAS References

In Givy we made the choice of using *raw C pointers* to reference objects in the GAS. More specifically, we define that each pointer returned by `malloc(size)` (and variants), on any node, references a block of memory `(ptr,size)` in the GAS. The pointer references the block at the GAS level (for transfers). And it also references the memory for the actual access in node virtual memory.

As a consequence, if a node requests access to a block of memory from another node, the block data must be copied to the requesting node local memory *at the exact position of its referencing pointer*. An example can be seen in Fig. 4.1, where blocks from node 1 will be copied at the exact same pointer in every node address space.

Figure 4.1: GAS layout example. The vertical direction represents addresses (pointers), the horizontal one represents the GAS address space, and the local memory states of 3 nodes.



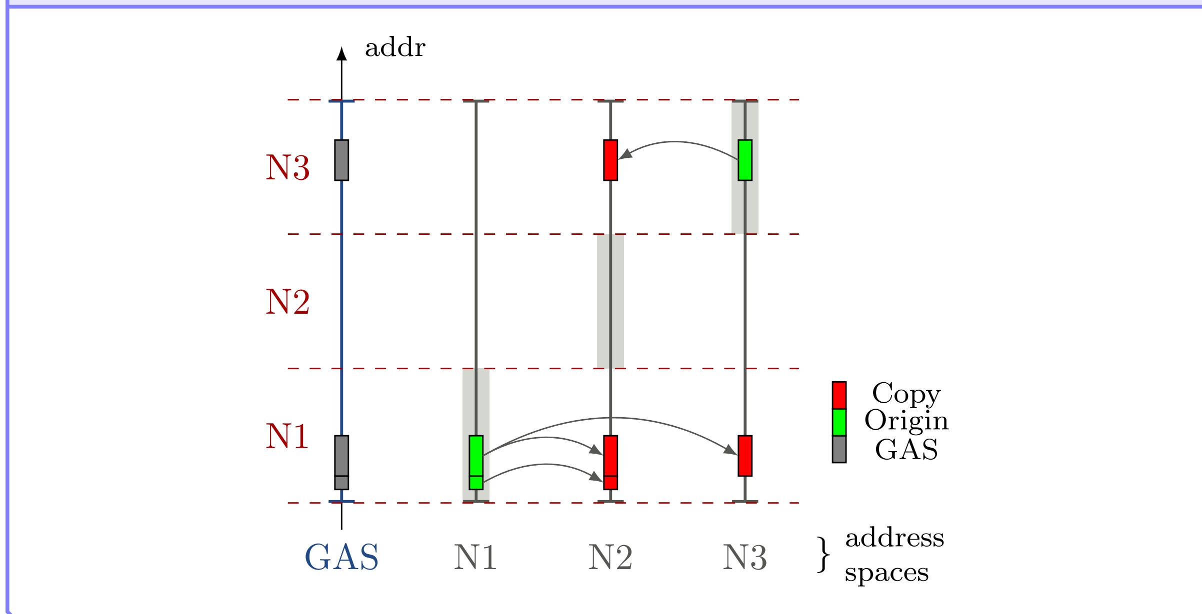
This choice is motivated by the availability of virtual memory on the MPPA [33], a MPSoC developed by Kalray. A goal is to have the portability of NUMA machines (which also supports raw pointers) with the flexibility of software GAS runtime systems (variable granularity, high level knowledge of application accesses). It removes the need for pointer translation when accessing data. RDMA network primitives can be used easily as we already know remote addresses in virtual memory. As a bonus, the MPPA architecture has virtual memory support in the DMA subsystem, eliminating the need for page locking.

`malloc()` Collision-free Property

Due to the choice of reference, each node `malloc()` allocates memory in both GAS and node virtual memory. Thus the set of node `malloc()` must create a coordinated GAS layout, with no collisions.

To prevent collisions, we split the GAS into *areas* (interval of pointers), and assign one exclusive area to each node (see Fig. 4.2). We then require each node `malloc()` to allocate inside the node assigned area. This strategy eliminates any risk of `malloc()` collision between nodes, and requires no network bandwidth for `malloc()` itself. `free()` may still generate some network messages for blocks which are not local, but it can be non-blocking.

Figure 4.2: GAS layout example with node areas. Node areas are indicated in dark red, and delimited by dark red dotted lines.



It does not put a hard limit on allocated memory: each area is an interval in virtual memory and can be huge compared to physical memory. Every node can still request access to pointers in any area.

Remote Block Memory Management

We decided against premapping the whole GAS on each node. It follows that before copying a remote block locally, we must ensure that virtual memory at its specific address is usable (mapped). As said in the description of virtual memory (Section 2.1), there is no method to quickly get the mapping state of virtual pages. Remapping blindly may destroy the current mapping and loose already present data. So we need a *fast mapping check* to decide when to `mmap()` part of the remote node areas.

> Premapping a huge chunk of virtual address space if popular in current allocators. It requires lazy physical page allocation, and forces the OS to reserve space for page tables. On the Linux kernel – which uses many lazy strategies in the virtual memory system – premapping is effective, but it may not work on embedded systems. It is probably best to abstract mapping management and switch between strategies depending of the machine.

Coherence Metadata Access

All coherence protocols use some kind of metadata, that must be accessible from the memory block reference. We decided against modifying user types by annotation and insertion of coherence metadata. So this metadata must be placed at a retrievable place automatically and silently. As the memory allocator of Givy also controls the memory layout of the GAS, we decided to let it control coherence metadata placement. Thus coherence metadata placement and access will be described in this section.

The internal area layout provides another critical property for the coherence protocol: it allows Givy to determine the allocation node of a region from the address only. The address must simply be matched against the area layout, which is the same on every node of the global address space. Moreover, the *creator* (allocation node) of a region is determined in constant time, with a node local operation.

4.3 Allocators – State of the Art

To the best of our knowledge, no current shared memory allocator provides our three required properties (collision-free, remote areas management, coherence metadata). Out of six memory allocators discussed below, only Streamflow and tcmalloc have page table like structures that could be used to implement our address space constraint. However these are internal structures that were not designed for fast lock-free read access, which is the requirement that lead to the design of the *Page Mapper Fixed* (PMF, Section 4.4.3). Redesigning them to allow lock-free accesses is usually not compatible with the rest of the allocator code base.

Hoard [21] is a high performance allocator written using Heap-Layers [22] which uses a C++ class pattern to create the allocator from modular components. It however lacks synchronization-free thread-local allocations and fast atomic remote-free, that both allow better performance in other allocators and may be critical on our target machine with very relaxed memory.

tcmalloc [9] is a more recent, fast allocator. It mixes blocks from multiple superpages (and thread heaps) in its freelists, using a garbage collection system to balance the thread heaps This makes controlling the page footprint difficult as superpages cannot be reclaimed easily. PMA quicklists are inspired by the *large object allocation* structure of tcmalloc.

jemalloc is the BSD default system allocator, written in C, with very good performance in speed and memory. Its implementation is mature (contrary to some research allocators), and has very high performance.

Streamflow [63] is a speed-oriented research allocator, that introduced the atomic remote frees and private heaps with synchronization-free thread-local allocations. The local allocator in Givy is heavily inspired from it, because these properties fit well the relaxed memory model of our target architecture.

SFMalloc [64] is a more recent allocator that has even less synchronization that Streamflow. It also removes page headers by packing them at the top of a huge page, which is good as it simplifies the memory layout. It also reduces the cache miss ratio due to searches in header lists (because headers are always at the same offset in pages, and hit the same cache lines). But it suffers from superpage fragmentation and from unstable

memory compactness. As a result, it does not seem well adapted to our local memory constraints.

SSMalloc [43] is a standard high performance allocator, but it solves the problem of page header cache conflict by forcing its memory chunks to not be aligned at page boundaries. This design gives it a speed boost, but its overall performance remains average.

4.4 Allocator V1

The first version of the Givy allocator was designed as a prototype. Its goal was to test the viability of a GAS allocator fulfilling the 2 constraints identified in Section 4.2:

- restrict `malloc()` operation to an interval
- provide a *fast mapping check*

Listing 4.3: Allocator V1 allocation primitives

```
void* malloc(size_t size) { /* impl */ }
struct coherence_metadata { /* ... */ };
void* gas_malloc(size_t size) {
  void* p = malloc(size + sizeof(struct coherence_metadata));
  init_metadata((struct coherence_metadata*) p);
  return p+sizeof(struct coherence_metadata);
}
```

The current Givy model places coherence metadata just before the memory block. This makes it very easy and fast to find this metadata from the reference (pointer to block), but can not support references to the inside of the block. However, before trying to access the metadata, we must ensure that encompassing pages are mapped, or map them if this is not the case. Thus, the requirement of metadata access becomes a *fast page mapping check*.

GAS Allocator Overview We now describe the structure of the Givy allocator system of one node, illustrated in Fig. 4.3. The *local allocator* (Section 4.4.5) behaves like a classic shared-memory multithreaded allocator, with the added interval property. `malloc()` (blue path) is truly local, while `free()` (red path) might send node-remote free requests if the region was shared, but will not block.

The *proxy allocator* (Section 4.4.6) manages the memory belonging to other node areas. `proxy_mapped(p, sz)` implements the *fast page mapping check* for pages containing the region `p`. `proxy_alloc(p, sz)` and `proxy_free(p, sz)` respectively prepare and destroy the virtual memory for further accesses. The proxy allocator mostly reacts to requests coming from the network or from the task scheduler (fast mapping check before looking at/creating the corresponding metadata). Incoming remote free requests will destroy the block copy (or original if we are the creator node of this region). The task

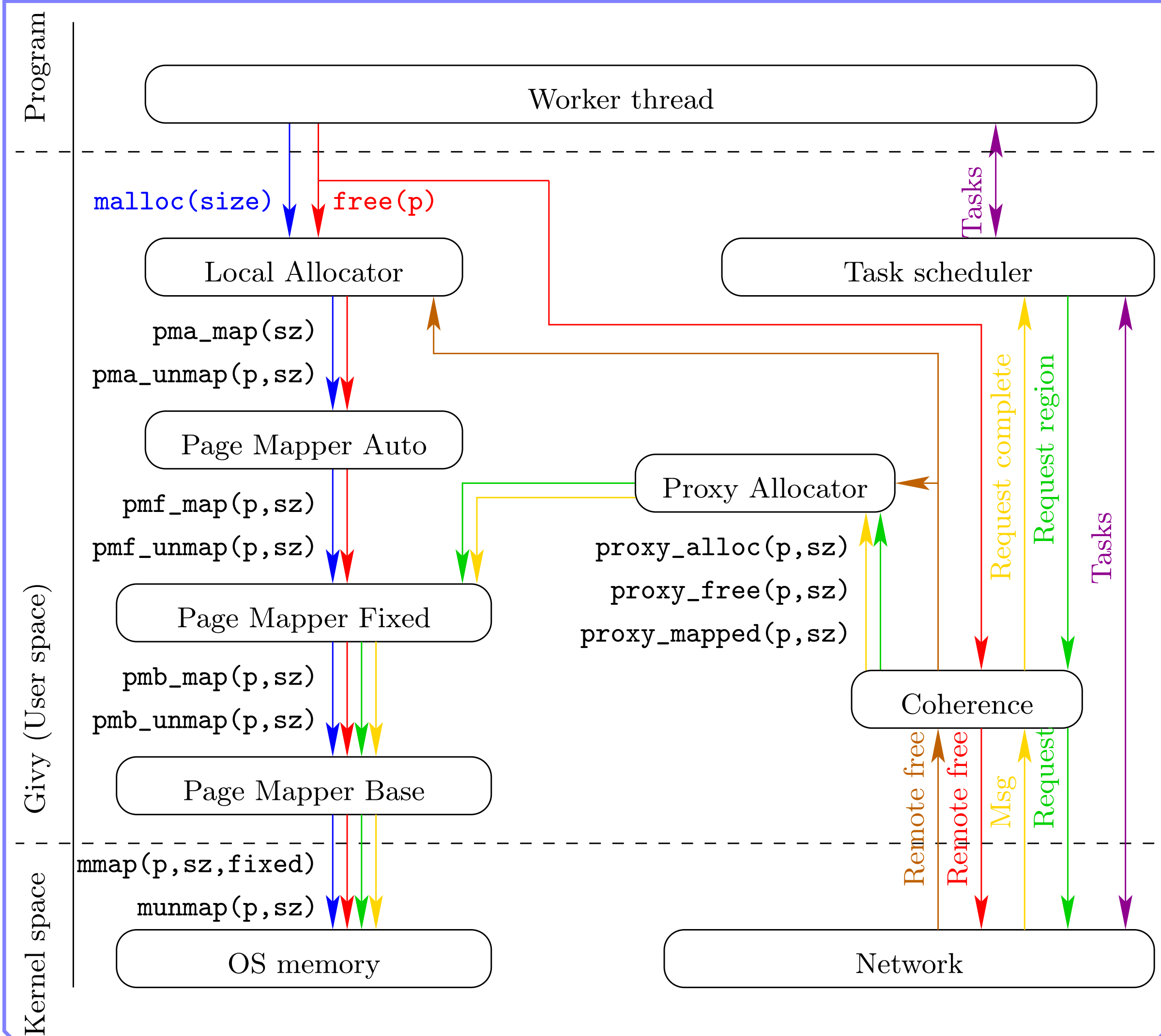


Figure 4.3: Givy overview with v1 allocator. Colored arrows represent possible call-graphs of specific runtime operations from their provenance. Bidirectional *Task* arrows indicates task scheduler specific interactions (not described). Grey function names represent the allocator structure interfaces (p=pointer, sz=size).

scheduler and coherence manager also talk through the network to implement the cache coherence protocol and migration of tasks, but this is outside the scope of this paper.

Both local and proxy allocator rely on the *page mapper* triplet of structures, described in detail in the next section. As a quick overview, the Page Mapper Auto manages automatic allocation in the node interval, the Page Mapper Fixed registers allocated memory and provides the fast mapping check, and the Page Mapper Base is a simple wrapper to the virtual memory interface.

4.4.1 Page Mapper

POSIX operating systems manage virtual memory mappings with the `mmap()` system call (old `brk()` is deprecated). `mmap()` allocates virtual memory from the system in units called *pages* (usually 4KB or higher). Classic memory allocators get pages with

`mmap()` and must fulfill user requests while being fast, avoiding false-sharing, and avoiding fragmentation.

So the control of `malloc()` addresses must be done at the `mmap()` call. The default behavior of `mmap(addr, size)` is to return any unused contiguous page sequence which size is greater or equal than `size`. `addr` is only a position *hint* and has no guarantee.

With the `MAP_FIXED` flag, `mmap(addr, size, FIXED)` forces the system to create a mapping *at* virtual address `addr`. However, if it overlaps with a current mapping the previous one is *overwritten without notification*. So we need to know the mapping state of the node virtual address space to find non-overlapping pages. The *page mapper auto* (PMA, Section 4.4.4) provides this information.

We also need to track the mapping state of other pages for the proxy allocator. Under POSIX, the only way to access this state uses system calls. As a consequence, Givy maintains a user-level data structure to speed up accesses to the mapping state, called the *page mapper fixed* (PMF, Section 4.4.3.

Some allocators like Scalloc [11] allocate huge amounts of virtual memory, relying on the lazy mapping strategies of current operating systems. This strategy is not desirable for embedded systems; OS mapping metadata takes space that may not be negligible for embedded systems. It can also lead to page fault and page mapping when testing for coherent metadata that is not present, which might be useless if the related task is finally scheduled elsewhere. It also complicates memory footprint management.

4.4.2 Page Mapper Base

The *page mapper base* (PMB) is a very small structure that wraps the calls to `mmap(addr, size, FIXED)`, or the system specific API for virtual memory. It also tracks the total number of pages taken from the system. All Givy memory mapping updates use it, so it enforces a memory limit on the whole runtime system. This is an interesting property for embedded systems; other dynamic runtime systems usually have unbounded memory usage.

4.4.3 Page Mapper Fixed

The fast page mapping check is the property with the biggest impact on task performance. It is part of the coherence metadata check which is on the critical path for task execution. Another use (but less performance critical) is to detect if pages are already mapped before mapping them using `mmap(addr, size, FIXED)` during a call to `proxy_alloc()`. Already mapped pages might hold other memory regions data, and must not be remapped by `mmap(addr, size, FIXED)` as it would destroy the current data.

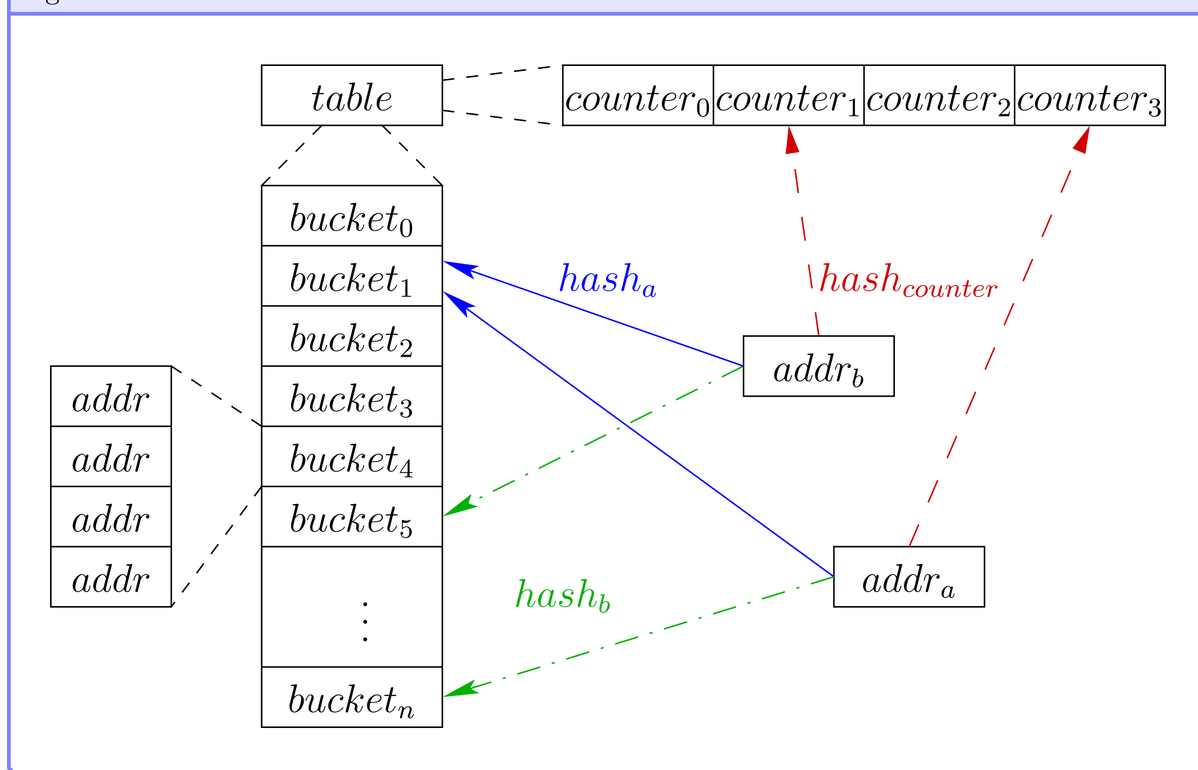
The virtual memory mapping is usually stored in the *page table*, an internal structure of operating systems which is not readable in user space. Checking the mapping state of a page can only be done by testing if systems calls like `mprotect()` return an error, which is too slow for Givy. So Givy keeps track of the state of memory mappings in a user-space structure similar to a page table called *page mapper fixed* (PMF). In the case of the MPPA, future work may allow to use the PMF as the system page table.

Hashed Page Table Most page tables use a fixed depth tree implementation [40], with very wide nodes (taking a 4KB page each). The Givy runtime could consume a lot of nodes in case of sparse mapping requests, which would consume way too much memory on the MPPA (which has only 512 4KB pages). Restricting the virtual memory space is not desirable: the host system with a huge physical memory may be part of the GAS, and must be able to map its memory.

Concurrent trees with lock-free accesses are complex and require memory management of nodes. So a better alternative is to implement the page table-like structure using a *hashed page table* (HPT), which stores the virtual memory mapping in a hash table structure. This also reduce memory usage compared to a tree implementation: on the MPPA (32bit virtual space/2MB physical memory per node), the table only consumes 2KB.

Concurrent Cuckoo Hash-Table The hashed page table is a *concurrent cuckoo hash-table* adapted from [36], which provides constant time lock-free lookups but needs serialized updates. As shown in Fig. 4.4 the table is cut into buckets that directly contain the stored keys (virtual page addresses in our case). It avoids using linked lists which require memory management. We cannot use more virtual pages than physical pages, so the table capacity can be set to fit the constant physical memory size, avoiding any resize operation.

Figure 4.4: Cuckoo hash table structure



Collisions are avoided by using two hash functions; each key must be placed in one of its two possible buckets. A third hash function maps to a cell in the small *synchronization counter array*, which is initialized to 0. Any modification related to a key first increments

the associated counter (from even to odd), performs the modification, then increments the counter again (from odd to even). Lookup of a key consists of reading the counter, checking if the key is in the buckets, then reading the counter again. The lookup must be started over if the counter changed or was odd to start with.

Without concurrent modifications, key lookup is constant time, and fast because each bucket fits into a cache line. Deletions are almost as fast as lookups. Insertions might take longer due to the possible swaps of key between buckets (cuckoo swaps), to create space if the two insertion buckets are full (see the original article for details [36]). However, as both types of table modification take place when a slow mapping modification (system calls) is done, this does not hurt performance.

Our implementation of this lock-free cuckoo hash table is built with the modern C11 atomic primitives for portability purposes (see Section 5.2.2). It only uses *release-acquire* and *relaxed* accesses (no atomic *read-modify-write*), which have the same cost as standard loads and stores on x86 [19]. All updates are serialized by a lock. The implementation has been verified on the CPPMEM model checker [3].

4.4.4 Page Mapper Auto

The *page mapper auto* (PMA) manages the allocation of *contiguous page blocks* inside the node area. Its interface is equivalent to a `mmap(addr, size)` constrained to the area. Its role is finding a sequence of unmapped pages in the area then using `mmap(addr, size, FIXED)` to map it. The hash table inside the page mapper fixed would require a costly linear search, so we need a dedicated structure. All PMA modifications must be sequential, and are protected by a lock.

The PMA views the area as a sequence of *page runs* (Fig. 4.5), which are sequences of consecutive pages with a common purpose. Each page run is in one of 3 states:

- *owned* (O): pages are mapped and given to the program;
- *cached* (C): pages are mapped but are unused by the program and kept to answer a future request;
- *unmapped* (U): pages are unmapped, this is invalid virtual memory.

The initial state is a completely unmapped area. All further operations will only perform cuts, merges, and state changes on this area and its subdivisions. This ensures that any page manipulated by the PMA will be in this particular interval, giving us the desired property.

It also makes caching of PMA's unused pages relevant, because they are in the right address interval and thus can be reused. The cache can avoid many system calls to `mmap()` or `munmap()`. All operation will merge cached (resp. unmapped) adjacent page runs to reduce fragmentation.

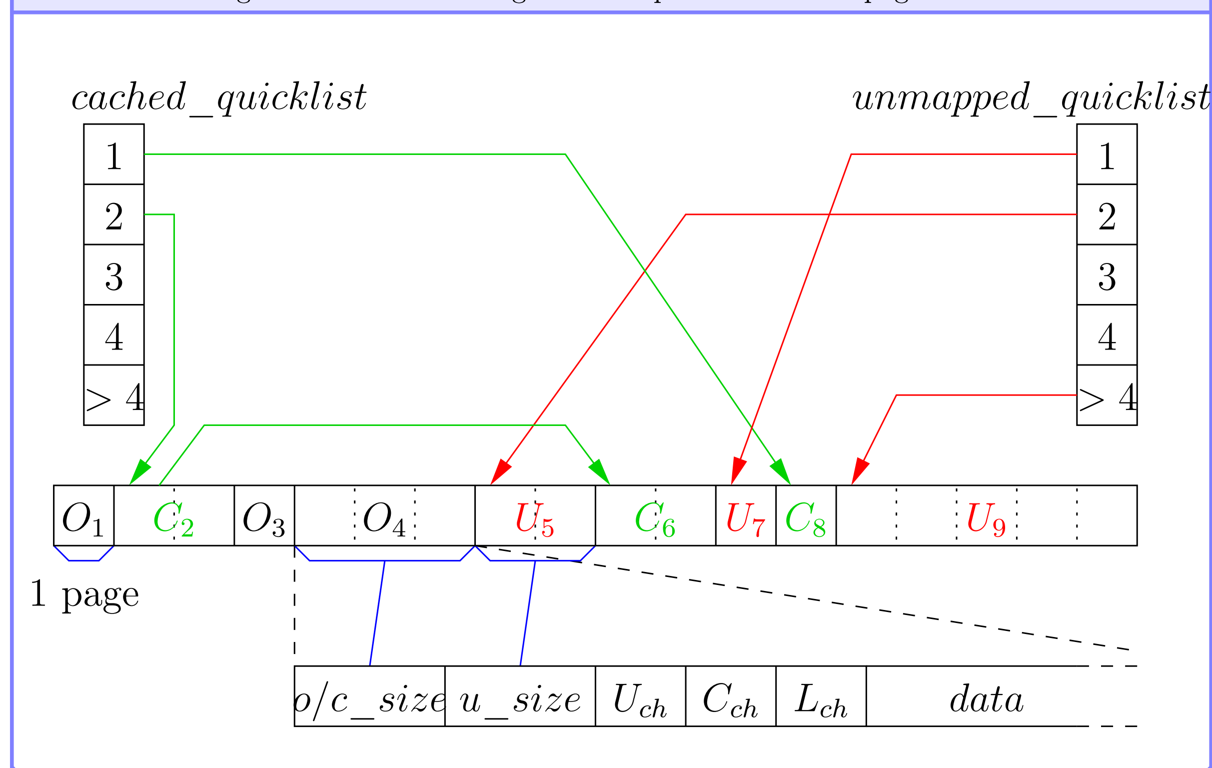
Page run requests are serviced quickly by using *quicklists*. The *cached quicklist* is a small array of N doubly-linked lists ($N = 10$ in our implementation). The k-th *fixed size list* contains cached page runs of k pages. An additional *large list* contains all bigger cached page runs ordered by increasing size. The *unmapped quicklist* has identical structure but contains unmapped page runs.

Page Run Allocation An allocation request for a k-sized page run will take the smallest page run of size at least k from the cached quicklists. If found, it it trimmed to k pages, given to the program, and the rest placed back into the cached quicklist. If not found, the same search is performed in the unmapped list, and the newly found unmapped page run is then mapped with `mmap(addr, size, FIXED)` before being given to the program.

An allocation is unlikely to fail due to fragmentation of the virtual address space, as the area is huge enough compared to the PMB memory limit. Most failures are due to reaching the PMB limit; in this case, we flush the cached pages (starting from the smallest to reduce fragmentation), then retry. The primitive which unmaps cached page runs can also be called from the proxy allocator if needed.

We illustrate these mechanisms with an example of PMA state in Fig. 4.5. An allocation request of 1 page gives C_8, 2 pages gives C_2. However, an allocation of 3 pages will find no suitable cached page run. It will pick U_9, cut and map the 3 first pages of it, and put the remaining 2 unmapped pages page run in the list. If the PMB limit was set to 12 pages, the mapping of 3 pages would have failed. The PMA would be forced to unmap C_8, that would merge with U_7 and U_9 to make a 7 page unmapped page run, which would be cut to give the 3 requested pages.

Figure 4.5: Example of *page mapper auto* data structure. Cached and unmapped quicklists are represented by arrows. $O/C/U$ stands for Owned, Cached, Unmapped page runs. Indices are for referencing. The bottom rectangle is a simplified view of a page run header.



Page Run De-allocation A page run de-allocation only consists in moving it to the cached quicklist. The page run is merged with neighboring cached page runs to limit cache fragmentation (in Fig. 4.5, de-allocating O_1 or O_3 will merge it with C_2). Page

runs will only be unmapped later if pages are needed, when the proxy allocator or the PMA will call the cache releasing primitive. It keeps the cache filled and does not cause any memory resource problem for external code because we comply with the PMB limit.

Implementation Details Each mapped (owned or cached) page run begins with a header. Unmapped page run cannot have a header, as their memory is unusable. Due to the merging, every unmapped page run is preceded by a mapped page run (or the start of interval). Thus we choose to store unmapped page runs information into the *preceding* mapped page run header. As shown in the example in Fig. 4.5, a header will store the size of the current cached or owned page run and the size of the following unmapped run (possibly 0). Although the PMA can work independently of the PMF on a machine which has an OS page table, we map/unmap pages through the PMF to simulate its role as a real page table in the future.

4.4.5 Local Allocator

The *local allocator* (LA) is the classic allocator front-end of the Givy memory system. It uses the page mapper auto to get pages inside the node's area, ensuring that any memory coming from `malloc()` is inside the area.

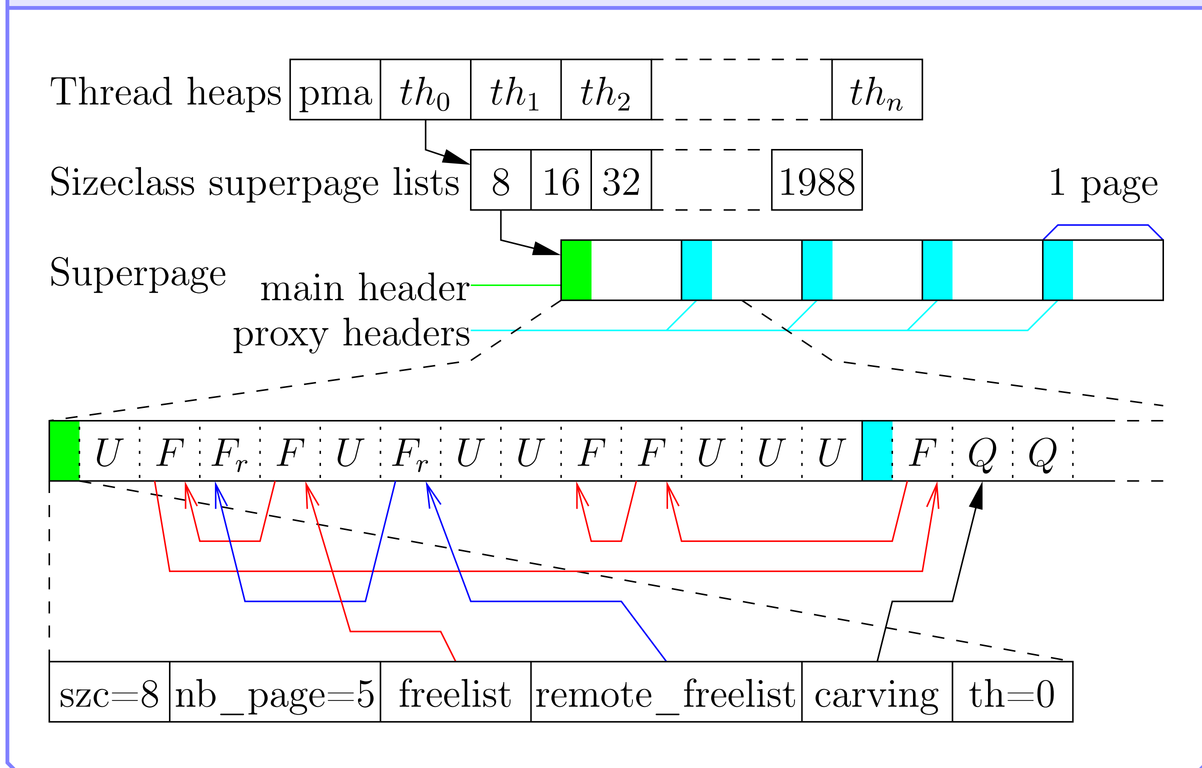
Most high performance allocators act as a memory cache between `mmap()` and `malloc()` requests, so we could adapt any current allocator to use the PMA. In practice, the PMA headers will interfere with allocator structures. Moreover, most allocators are either production quality, with a big codebase providing profilers and extended APIs; or research quality, with a focus on speed for big allocations at the expense of memory usage. Either way, these allocators are usually not adapted to small embedded systems with only 2MB of memory. Thus we designed a light multithreaded allocator that is similar to Streamflow [63], but tuned to fit the small memory and compatible with the PMA.

Design As a general rule, every page manipulated by the local allocator is freed as soon as possible. No page caching is done at this level. We rely on the efficient PMA page run caching, which however is protected by a lock to enforce the sequential modifications. All allocations above half a page size are rounded up to a multiple of page size and allocated directly from the PMA. A small header is added to indicate that this is a *big block*.

For smaller allocations, we use the structure illustrated in Fig. 4.6. To release contention on the allocator, smaller allocations are serviced from a synchronization-free private *thread heap*. Worker threads in the Givy runtime are supposed to live as long as the program, so support for thread heap reclamation is not required and has not been implemented. This heap is composed of *slab allocators* — allocators that only allocate a fixed size — for each size (called *size class*) in a predefined list. The allocator will choose the nearest size class that fits the allocation request and return a block from the corresponding slab allocator.

For efficiency, each slab allocator gets its memory from the PMA in chunks called *superpages* (a mapped page run). Each superpage is cut into sizeclass sized blocks, that are managed by a superpage local header (this allow to easily detect a completely unused superpage for reclamation). This header has a *freelist* that links all unused blocks freed by the local thread. A *remote freelist* [63], accessed with atomic operations, links

Figure 4.6: Structure of the local allocator (see Section 4.4.5). $U/F/F_r/Q$ represents Used, Freed, Remote Freed and Quarry blocks.



unused blocks freed by other threads. A *carving index* indicates where the next block can be created in the never-used part of the superpage (*quarry*). When allocating from a superpage, we first try to use the freelist, then remote freelist, then carving, failing to allocate if the superpage is fully used.

A slab allocator contains a list of superpages that are tried in order when allocating. Recently used superpages with free space are moved to the front to reduce allocation time. If all superpages are full, a new one is obtained from the PMA.

When freeing a block, we look for a header at the start of the page containing the block. This header is either a big block header (we call the PMA), or a superpage header, or a *proxy header* that redirects to the superpage header. If we own this superpage, we simply add the block to the freelist, and return the superpage to the PMA if it is completely unused. If the superpage is owned by another thread, we use an atomic operation to add the block to the remote freelist.

Small Memory Tuning In the slab allocator, space is wasted on each allocation if it doesn't fit exactly the sizeclass (*internal fragmentation*, scales with memory usage); and if the superpage is not completely full (*superpage fragmentation*, is amortized when memory usage increase). Many allocators use a lot of close sizeclasses which is better for big memory consumption scenarios (small internal fragmentation and amortized superpage fragmentation). In our case, superpage fragmentation is more important than internal fragmentation, so we use fewer sizeclasses, based on a power of 2 sequence. This has the effect of clustering more varied allocation requests per virtual page, which also helps

the proxy allocator as remote copied blocks are more likely to be in the same page. We provide a way to define application specific sizeclasses to let the application remove internal fragmentation on some sizes, if needed.

In Streamflow [63], each superpage contains at least 2000 blocks of their sizeclass, which can create superpage fragmentation. In our design, superpages can be created in different sizes, that grow for each allocation of this sizeclass (with a cap). This allow for a more gradual memory usage.

4.4.6 Proxy Allocator

The *proxy allocator* manages the memory areas assigned to other nodes, based on functionality provided by the Page Mapper Fixed. It attaches a reference counter to each page to keep the current number of region copies inside the page, and detect when to unmap the page. Calls to `proxy_alloc()` will map the encompassing pages or increment the counter, and rely on the page mapper fixed to detect if pages are already mapped. This counter is placed where the local allocator would place its metadata if the region was allocated locally. This space is guaranteed to be unused because each remote region copy has originally been allocated by an instance of the local allocator.

4.4.7 Measurements

Performance evaluation of Givy's allocator subsystem is a complex problem. Measurements on executions with the whole Givy GAS distributed runtime would give little to no insight on the allocator subsystem performance. The performance of a distributed GAS runtime depends mostly on:

- The performance of the network stack (hardware and software), as each GAS cache miss will trigger multiple slow network transfers compared to a GAS cache hit (which only requires a local metadata check).
- The design of the GAS cache coherence protocol, which controls the number of slow network messages.
- The quality of the data and computation distribution across the computing nodes, which impacts the ratio of GAS cache hit versus miss.
- The performance of node-local facilities like malloc and other used libraries.

So measurement on the whole runtime, and possible comparison to other runtimes, would give information on the network performance, cache coherence protocol design, and scheduler capabilities, which is not the subject of this paper.

We decided to test the allocator subsystem in isolation to be able to interpret its performance. We designed several benchmarks to test the multiple features of the allocator subsystem: (1) the local allocator malloc implementation; (2) the proxy allocator that manages the local address space for remote region copies; (3) interference between local and proxy allocator.

Experiments have been done on an AMD Opteron 6164 HE (12 cores, 800MHz), as the target MPSoC (MPPA) lacks a full virtual memory support. They are compiled with gcc-4.9 in C11 mode (to support C11 atomics). Time measurements used

linux available timers. Memory measurements used memory high-water marks available in `/proc/self/status`. Coherence metadata access has not been benchmarked as no meaningful comparison exist.

Local Allocator Performance Evaluation

The local allocator is the implementation of a GAS distributed allocator. Few other GAS allocator exists, and most generate network transfers contrary to ours, making comparison difficult [46]. Each node instance of the local allocator is independent from all others, and behaves like a classic shared-memory multithreaded allocator. So we choose to compare one node instance of our allocator against state-of-the-art shared memory allocators, to determine if providing our property degrades performance. Our allocator was compiled into a small shared library which overrides the `malloc()` API symbols, for use with `LD_PRELOAD`. The PMB memory limit was set to two times the expected memory load to prevent spurious failures.

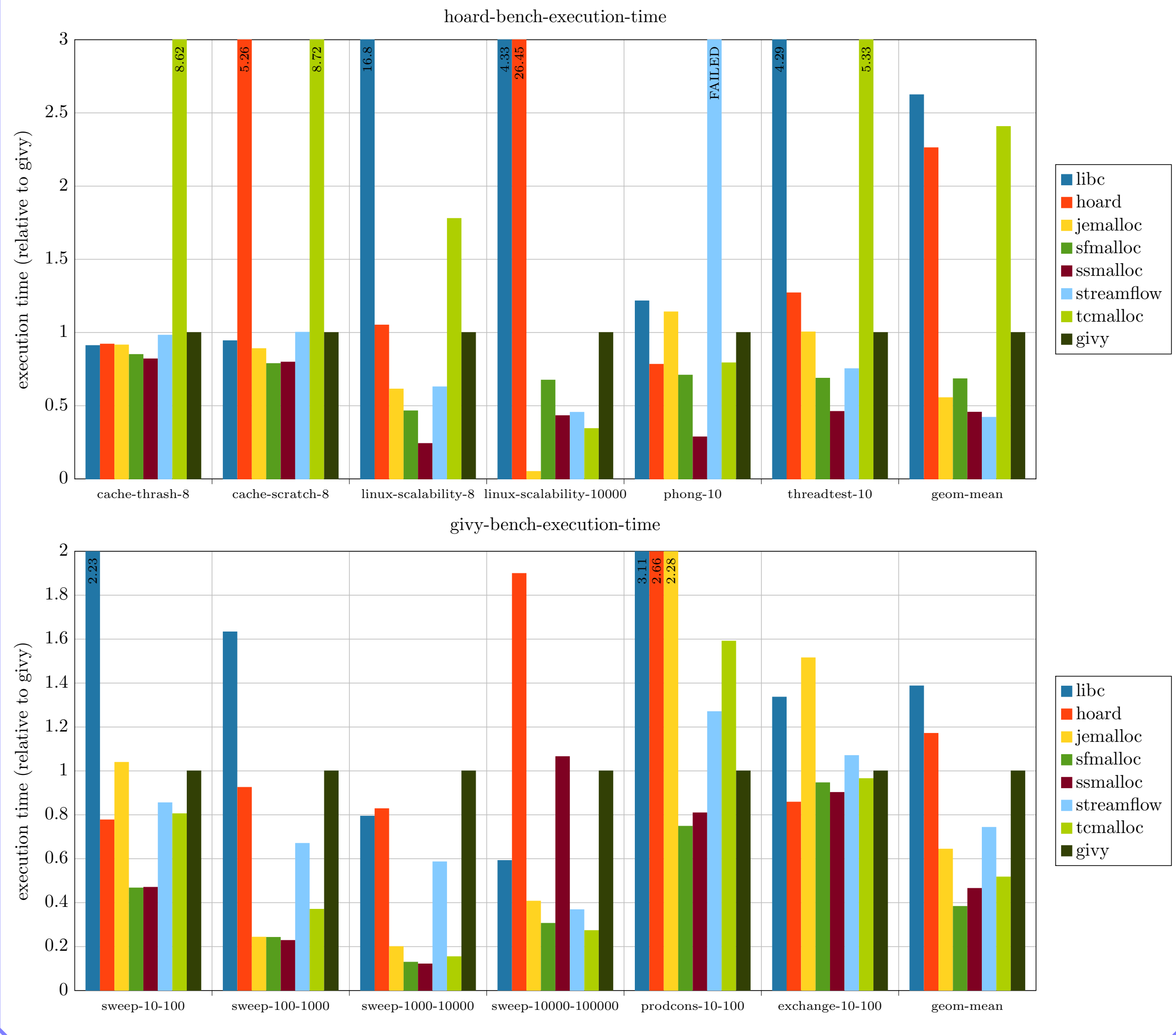
Our experiments check both allocation speed, and memory compactness which is important for embedded systems. We first used synthetic reference benchmarks from Hoard [21]. We excluded the Larson benchmark, as it continuously creates and destroys threads, which fails due to our allocator lacking thread heap reclamation (Section 4.4.5).

We also added more synthetic benchmarks to detect performance anomalies: (1) ThreadTest stresses the allocator synchronizations as lots of calls to malloc and free are executed in parallel for an object size (denoted as ThreadTest-*size*). (2) To evaluate fragmentation, we did also develop a variation of ThreadTest where object sizes are randomly selected in a range (denoted sweep-*range*). sweep consists of many consecutive phases of parallel allocations and de-allocations, separated by a synchronization barrier. Phases begin and end without any allocated objects and are separated by a synchronization barrier. (3) Then, to evaluate the ability of the allocator to transfer unused memory from one thread to another, we developed a new synthetic benchmark called exchange-*range*. exchange is similar to sweep, but contains only two threads and when one is allocating objects, the other one is de-allocating (his own) objects. (4) A last synthetic benchmark called prodcons-*range* evaluates the performance of thread remote-free. prodcons is similar to exchange but the thread de-allocates the objects allocated by the other one instead.

However they are not realistic enough to validate our approach as they only perform allocations/de-allocations unlike real applications which spend part of their time performing actual computation. Finding interesting real benchmarks is not easy in our context: embedded system benchmarks rarely stress allocators, as they tend to preallocate data before computation. OpenStream [61] is a shared memory task runtime, and thus is similar to the environment where our allocator will be used; it is our most realistic benchmark. A task runtime might generate frequent calls to the allocator to create and destroy the task data structures, and is thus very interesting to check allocators performance.

Local Allocator Speed We compared our allocator with seven different state-of-the-art existing allocators described in Section 4.3: libc, Hoard, jemalloc, tcmalloc, Streamflow, SFMalloc, and SSMalloc. The goal of these experiments is not to perform a pure comparison between allocators but to detect possible issues with the design. As outlined earlier, those synthetic benchmarks are not realistic as actual applications do spend time

Figure 4.7: Execution time (relative to Givy) for Hoard and Givy benchmarks.

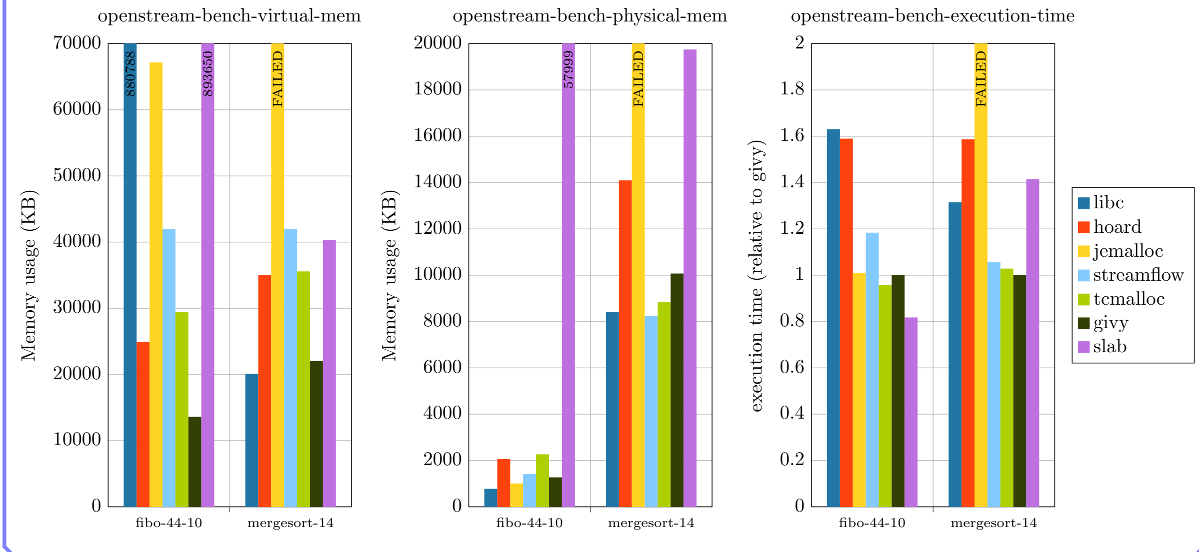


performing actual computations in addition to allocating and de-allocating data.

Fig. 4.7 shows the results for execution time (normalized to the one obtained with our allocator Givy). As expected, Givy behaves as well as the other state-of-the-art allocators in terms of pure performance. In particular we expected contention on the PMA to induce a slowdown for big objects. We do measure a small degradation of performance for sweep-10^4-10^5 just as Hoard and SSMalloc. As expected, Givy handles remote-free efficiently as shown in prodcons. cache-* benchmarks from the Hoard suite have been designed to evaluate the effect of false-sharing. tcmalloc behaves, as expected, pretty bad on this test, but strangely Hoard too, and Givy performs well.

OpenStream Benchmarks To evaluate Givy on more realistic application schemes, we selected two benchmarks from the OpenStream benchmark suite, and ran them with different allocators including Givy. Note that by default OpenStream uses a thread-private custom slab allocator to allocate internal runtime objects. This default configuration is called slab in benchmarks; for every other allocator, the slab allocator is disabled and

Figure 4.8: Virtual/physical memory footprint (in KB) and execution time (relative to Givy allocator) on OpenStream benchmarks.



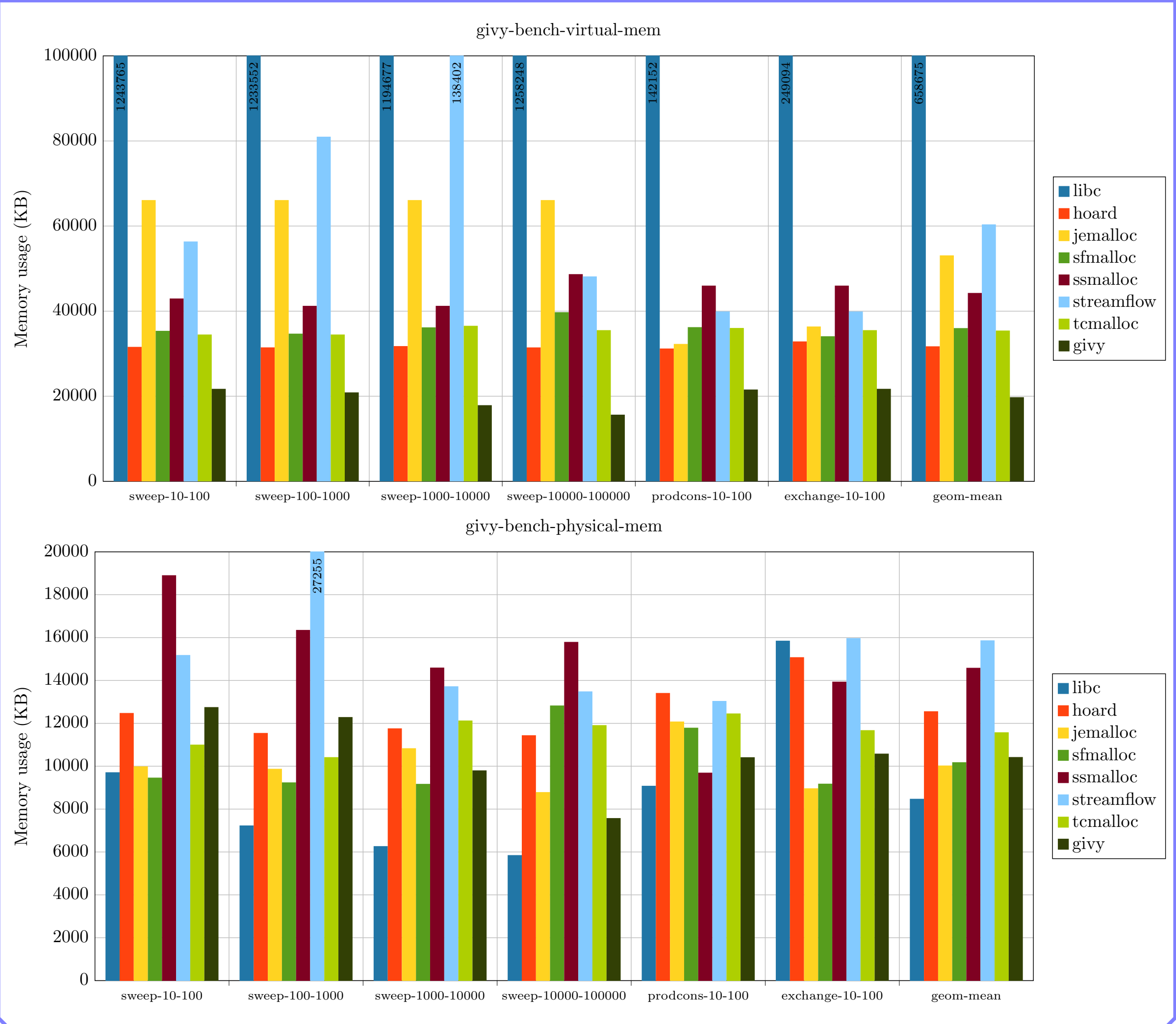
redirects to `malloc()`.

Fig. 4.8 shows virtual and physical memory footprint and execution time measurements for two OpenStream benchmarks. `fibo-44-10` is a recursive `fibonacci(44)` computation using tasks. 10 is a *cutoff* parameter — it indicates the value where we stop generating tasks and compute sequentially — that has been tuned to generate many tasks and stress the allocator with task frame allocations. `mergesort-14` is a task recursive mergesort of a an array of size 2^{14}. Results are coherent with synthetic benchmarks; Givy performs very well in virtual memory, has correct performance in physical memory and speed (execution time). Givy is better than the custom slab in almost every aspect; slab has very bad physical memory performance, probably due to lack of sharing mechanisms. slab also suffer from big virtual memory footprint; it is due to using libc internally and inheriting libc big footprint. jemalloc, crashed on `mergesort-14`, while SSMalloc and SFMalloc crashed on both and have been omitted.

Local Allocator Memory Usage The goal of the following measurements is to evaluate the ability of Givy to work in an environment with low available memory. For this purpose we measured both the virtual and physical memory footprint (max amount of used memory) of our eight different allocators. The overall allocated memory (per phase) was fixed to be approximately 10MB for sweep and 2MB for exchange and prodcons, similar to the amount available in our target architecture MPPA. The memory footprint is always higher as it includes mappings for the code and stack, which are not subtracted as it is part of the real memory footprint of the program.

Results of measurements of virtual/physical memory footprint are reported in Fig. 4.9 and in Fig. 4.10. Due to some Hoard benchmark not accessing the allocated memory at all, physical memory is not allocated for the virtual memory, explaining the big gaps in these benchmarks. Givy benchmarks always access memory, so they are more accurate to compare virtual and physical footprint. Givy allocator physical memory performance

Figure 4.9: Virtual/physical memory footprint (in KB) for Givy benchmarks (+geometric mean).



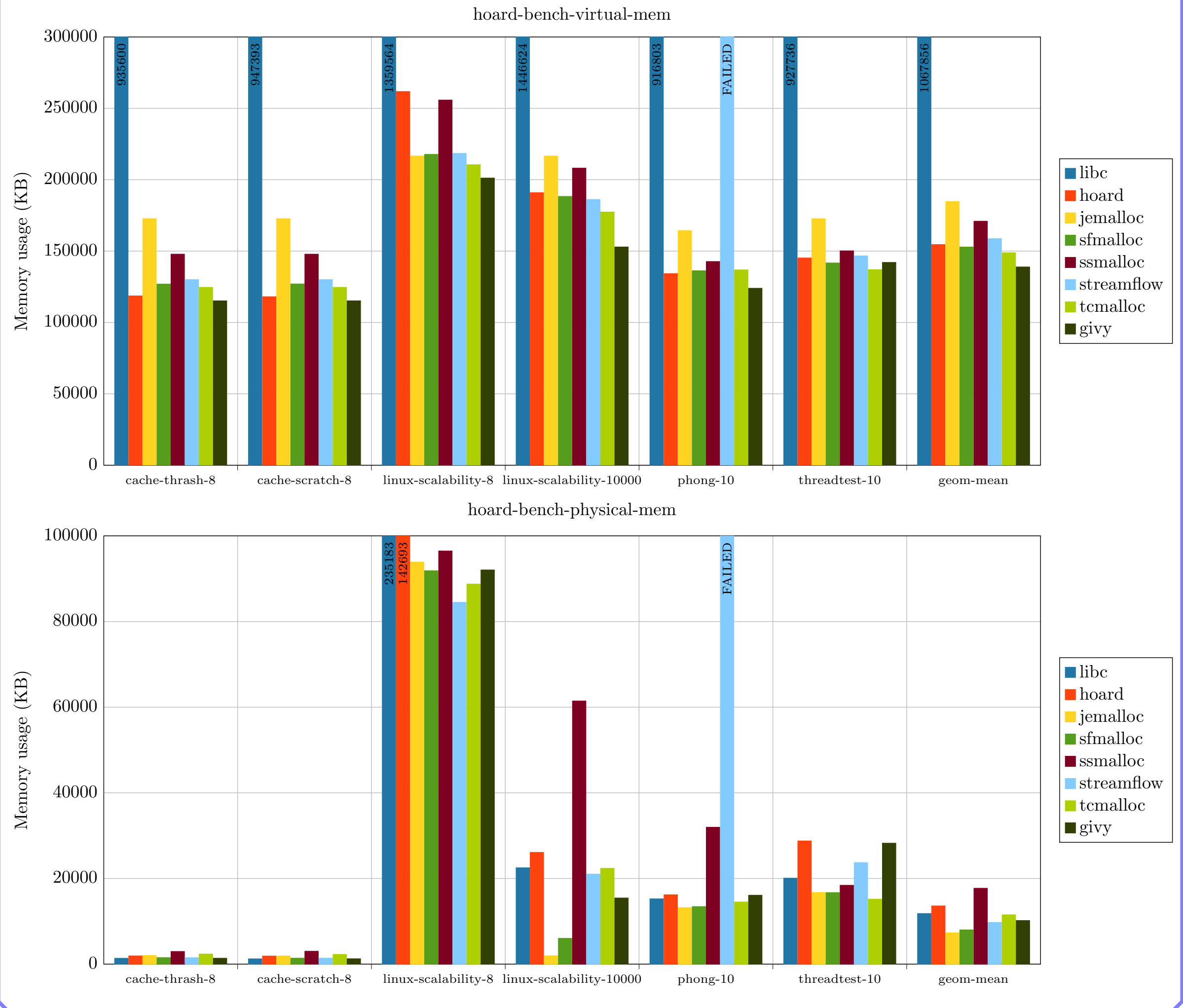
meets the low memory footprint. On average, only libc is significantly smaller, but suffers from huge virtual memory usage. Givy tightly controls its virtual memory footprint which is the lowest.

Overall, classic allocator front-end of Givy does not lose speed performance to provide its additional property. The reason is that most of the front-end speed performance comes from strategies such as thread local heaps or slab allocation, and not from the infrequent page mapping update where our property is enforced. Its memory performance is correct and is compatible with a non-lazy support of virtual memory.

4.4.8 Proxy Allocator Performance

The proxy allocator interface is different from `malloc()`/ `free()`: it takes a memory location (`(ptr, size)`) as input. This means it cannot be tested by a usual allocator benchmark; and besides the region sizes, a proxy allocator benchmark also needs to provide their previously determined memory addresses.

Figure 4.10: Virtual/physical memory footprint (in KB) for Hoard benchmark suite (+geometric mean).



The chosen benchmark runs some waves of `malloc()`/ `free()` and one wave of `proxy_alloc()`/ `proxy_free()` in parallel for a set duration, and measure throughput of both operations. The goal is to compare proxy allocator to local allocator operation time, and also to see if performance degrades when both run at the same time. The number of threads doing local allocator operations, the size of the wave, the size of allocated blocks are parameters. A slowdown loop allows to spread proxy allocator operations in time, reducing pressure and interference on shared structures. Only one thread performs proxy operations, because in the expected usage scenario (GAS runtime), only one thread performs them (the thread managing the coherence protocol). The sets of locations for proxy allocations are: (1) successive memory blocks, without space in between (*successive_ dense*); (2) successive memory blocks, with a fixed spacing between them, to test sparse proxy allocator requests (*successive_ sparse*); (3) matrix tiles from a Cholesky decomposition (*cholesky*); (4) matrix tiles from a QR decomposition (*QR*).

The matrix tiles locations are traces extracted from parallel executions of Cholesky and QR decomposition in the shared-memory OpenMP task runtime. The traces are the

Figure 4.11: Benchmark of `proxy_alloc()` / `proxy_free()`. `nb_threads` represents the number of threads running local allocator operations. 11-s indicates a benchmark with a slowed-down proxy allocator thread. Results are in allocator operations (`malloc()`/ `free()`/ `proxy_alloc()`/ `proxy_free()`) per second, averaged per thread. Higher is better.

	block_size	32B				512B				8KB			
	nb_thread	0	1	11	11-s	0	1	11	11-s	0	1	11	11-s
local	successive_dense		13.8M	3.6M	3.5M		9.1M	1.7M	1.8M		802K	28K	25K
	successive_sparse		19.8M	4.7M	3.5M		8.8M	1.7M	1.7M		791K	28K	30K
	cholesky		22.7M	5.1M	5.7M		8.8M	1.7M	1.8M		832K	26K	28K
	QR		23.3M	3.6M	3.6M		8.8M	1.7M	1.8M		815K	26K	27K
proxy	successive_dense	11.4M	6.9M	9.6M	521K	2.1M	1.8M	825K	445K	192K	156K	33K	19K
	successive_sparse	1.4M	1.1M	765K	332K	1.3M	1.1M	535K	340K	194K	156K	36K	31K
	cholesky	8.9M	8.4M	6.6M	519K	1.1M	982K	435K	306K	140K	119K	31K	30K
	QR	10.3M	9.7M	8.3M	526K	1.2M	1M	468K	305K	140K	120K	32K	27K

sequence of memory blocks of matrix tiles that each thread accessed. Although this is not a trace of a distributed memory task runtime, we think that each thread accesses represent roughly what each node would access in a distributed runtime. To respect the semantics of the proxy allocator (which can only manage blocks allocated from one local allocator instance), memory block pointers have been converted to pointers given by a local allocator instance. To have a meaningful comparison, the matrix tile size and trace size have been chosen to be similar to the *successive* benchmark wave size and block size parameters.

Results can be seen in Fig. 4.11. We first discuss the performance of the proxy allocator only, seen by looking at the column with `nb_thread` = 0. As expected, for small sizes (32B and 512B), the proxy allocator performs better if the requests are densely packed in memory (successive_dense vs successive_sparse). In such cases, multiple requests will fit into one page, amortizing the cost of mapping the page. For larger sizes, performance is equivalent between sparse and dense as any request will trigger page mappings. Surprisingly, cholesky and QR traces behave like dense for 32B, but more like sparse for higher sizes. This is probably due to cholesky and QR having sparse requests, but at 32B many of them are still bundled into a few pages, making it behave like the dense benchmark.

The performance of small sizes has a high variability — from 20 times slower to 3 times faster than the local allocator, if sparse or dense — which is due to high performance in small sizes of the local allocator, and high sensitivity to sparseness of the proxy allocator at these sizes. However, real applications in a GAS task runtime should try to use the GAS mostly for bigger structures, and pass small values as task parameters. In that case, the proxy allocator performance is more stable, between 8 times slower to 1.5 times faster. Overall, the proxy allocator is within an order of magnitude of the local allocator performance for a similar size.

We can also look at performance interference between the local allocator and proxy allocator. Except for successive_dense at 32B, increasing the number of local allocator operations threads decrease the proxy allocator performance. This was expected as both structures share a lock for access to the page mapper fixed. We can also see that at the maximum thread number of our machine, further slowing down the proxy allocator by the slowdown loop has no significant impact on the local allocator performance, especially for bigger sizes.

4.4.9 Conclusion

The version 1 allocator system was useful as a prototype. It helped identify the 3 core functionalities required from a GAS allocator subsystem : being collision-free at GAS level, remote block area management, and coherence metadata access. It fulfills all properties, and has adequate allocator performance for our target systems.

However it is very specialized for this target architecture. Its simple design does not scale well for gigabyte memory footprints that are common for X86 clusters. Moreover, the coherence metadata access is not implemented in a very efficient way. It creates coherence metadata for all allocated memory blocks, even if they are never shared, which wastes memory.

4.5 Allocator V2

The version 2 of the Givy allocator system improves on the first design to correct its deficiencies : high memory footprint performance, and better coherence metadata integration. It is implemented in C++ (like the version 2 of the runtime), and is less specialized for small memories, but can be configured for them. The *local allocator* part is complete, but coherence metadata placement and proxy allocator code are not (however their design is mostly done). Design is summarized in Fig. 4.12.

4.5.1 Basic Allocator Structure

Like modern allocators, the version 2 allocator manages the virtual memory at a coarse granularity. Version 1 allocator had *page runs* which could be split in *small blocks*. Version 2 allocator has *superpage blocks*, which can be split in *page blocks* which can again be split in *small blocks*. This deeper memory management hierarchy helps amortize allocator overhead, and reduces interleaving of allocator metadata and program data.

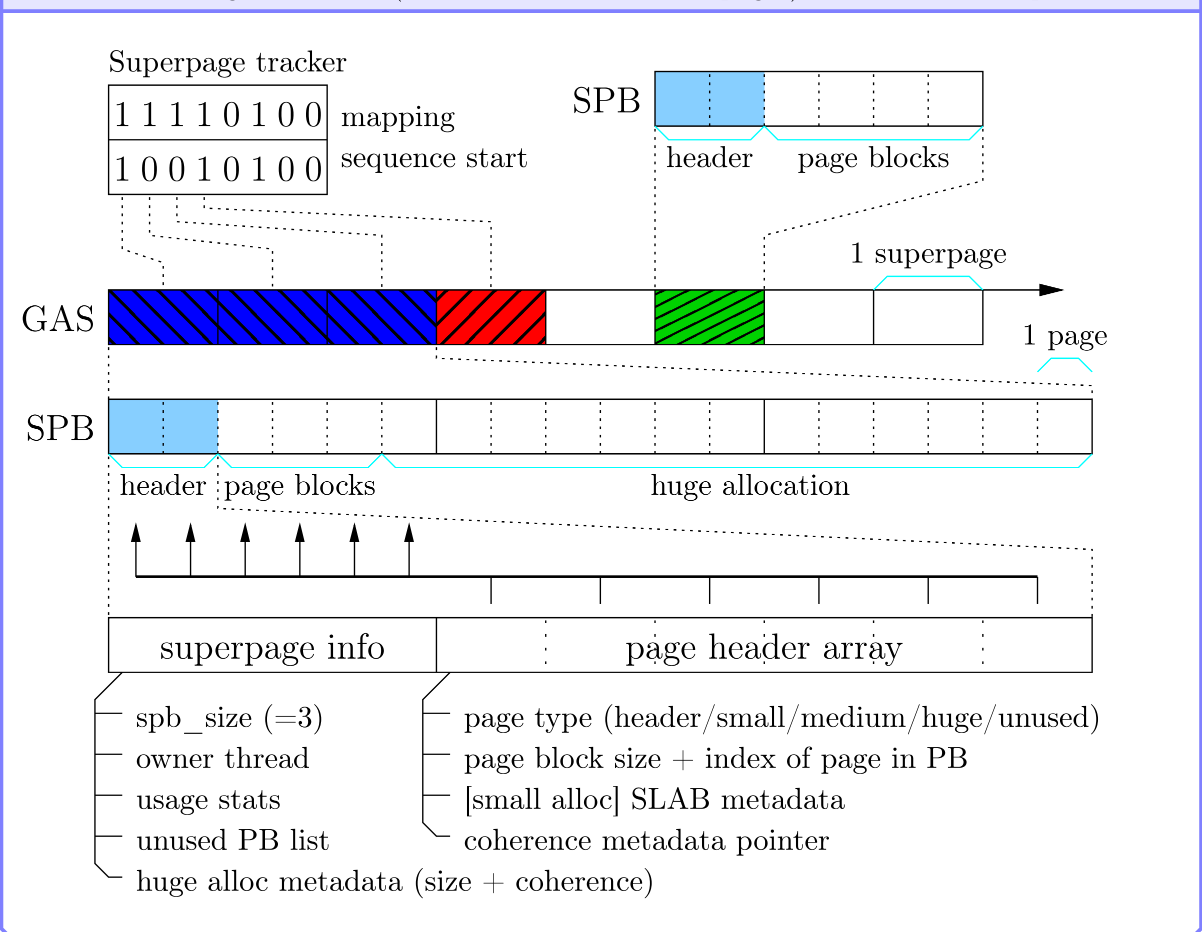
Superpage blocks (SPB) are contiguous runs of *superpages*, whose size is configurable at compile time to fit the target architecture. On targets with hardware superpages, the hardware superpage size should be used as the allocator superpage size. *Page blocks* (PB) are contiguous runs of *pages*, which should have the size of hardware pages. Using hardware page sizes as allocator management sizes let the allocator issue mapping requests with the same granularity as what the hardware supports. The allocator version 2 is written in modern C++ (C++14), and heavily uses features like *constexpr* to compute values like sizeclasses, offsets, sizes from a few configuration constant values.

Allocations requests are split into three categories: small (under page size allocations), medium (between page size and superpage size), huge (bigger than superpage size). A huge allocation will always use a superpage block of more than one superpage, and use the end of the superpage block for the allocation (rounded up to the next page size). Remaining page blocks of the superpage are added to unused page block lists. Medium allocations are fulfilled by returning entire page blocks, either from the unused space of superpage blocks from huge allocations, or from fresh ones with only one superpage. Small allocations use segregated slab allocators operating on page blocks.

The first page blocks of a superpage block are used to store all allocator metadata of the superpage block: metadata for the page blocks, their split layout, metadata for inner

small block management. No metadata is stored in any other page block of the superpage block, increasing data locality of both user data processing and allocator primitives. Page blocks are managed in a similar manner as *page runs* of the version 1 allocator: they can be used or unused, neighbouring unused page blocks are merged, a quicklist stores unused page blocks by size. Page block management is local to the superpage, avoiding fragmentation and allowing the allocator to easily detect empty superpage and unmap them. Each superpage is owned by a thread which can exclusively allocate from it. A *thread local heap* is associated to each thread (through a `thread_local` variable) to store the thread owned memory. The thread heap references all superpages owned by a thread. It has a quicklist that references non full page blocks used for small allocations: this increases small allocation speed. Thread remote frees are atomically added to a queue in the corresponding thread heap, and is flushed when the thread uses any allocator primitive. Superpages are orphaned on thread death, and will be adopted by the first thread that calls a primitive on them.

Figure 4.12: V2 allocator internal structure. Shows two superpage block (SPB) layouts: with and without a huge allocation (not to scale in number of pages). Thread local heap not shown.



4.5.2 GAS Memory Management

Tracking of the state of the virtual memory is now done at superpage level, thus modifications should be less frequent than in the version 1 allocator. The cuckoo hash table has been replaced by a lock-free bit array, called *superpage tracker*. This table tracks both mapping state (for *fast mapping check*) and superpage block layout (for superpage block allocation). It uses one bit table for mapping state, covering $2MB \times 4KB \times 8 = 16GB$ of virtual address space for each 4KB page used for the bit map. Another bit table is used to store superpage block layout: a 1 bit indicates the start of a new block. Checking the mapping only requires an offset computation and a single bit check.

4.5.3 GAS Coherence Metadata

This section describes the integration of coherence metadata in the version two of the allocator. This has only been partially implemented.

Compared to the first allocator version, coherence metadata is now segregated from used data, and created lazily for medium and small allocations. Huge allocations have a space reserved in their superpage header for one coherence metadata structure. Medium and small allocation access their metadata through a pointer in their corresponding page block header. This pointer is initialized to `nullptr`. When coherence metadata is first requested, it is dynamically allocated and the pointer set to it. Medium allocations page blocks allocate a single coherence metadata `struct`. Small allocations page blocks allocate an array of coherence metadata `struct` (one per small block in the page block).

Like the version one allocator, the version two has a *local allocator* and a *proxy allocator*. The local allocator manages the GAS interval owned by the node in the way described above. The proxy allocator manages other intervals. This version two design *requires* that some allocator information is transmitted along with coherence messages: type of allocation (small / medium / huge), and either sizeclass (for small) or position of page block (for medium). The proxy allocator also manages memory mapping at superpage granularity. It uses the same header layout as the local allocator: superpage header containing an array of page block headers. Proxy allocator headers can then be used in the same way as for the local one to access metadata for remote nodes. These headers are kept up to date using coherence messages which now contains allocator information.

A first good property of this design is reduced memory overhead due to coherence metadata. Small blocks are often used for temporary allocations (string manipulation, ...) and thus are unlikely to be shared in multiple nodes compared to huge allocations with data to be processed in parallel. In the version one allocator, they suffer for a lot of overhead as they all have a preallocated coherence metadata block. In this version, no metadata will be created if they are effectively not shared between nodes.

This version also supports metadata access from any pointer inside a block. The allocator already has to be able to find block metadata from any pointer inside (for `free()` when memory was allocated with a naive `posix_memalign()`). This design just uses the same system but accesses the coherence metadata instead, as it is managed by the allocator too.

The *memory region splitting extension* (Section 2.4.1) is much simpler to implement in the version two allocator. The segregation between data and coherence metadata allows to modify the metadata (creating a sub array of metadata for split) without changing the

data layout. Splitting generate requests associated to pointers inside the initial block, which is now supported.

The memory placement of dynamically allocated coherence metadata is still undecided. A first option is to use the local allocator itself, which requires no specific modification to the allocator design. It has the downside of mixing metadata with some data (usually small blocks). Another alternative is to allocate in a special *area* which is out of the GAS. The downside is to at least create one superpage which may be underused for small memory footprint applications, increasing the memory overhead.

4.5.4 Conclusion

Compared to the version one allocator, the increased virtual memory management granularity should help with allocation speed. However for small memory usage, this increased granularity might lead to bigger overheads (underused superpages). This overhead is in virtual memory, so operating systems could use lazy physical memory allocation at smaller granularity and mitigate the actual memory overhead.

The granularity is also configurable at compile time. However it must be the same on all nodes of the GAS, to let the *local allocator* and *proxy allocator* use the same layout. Givy and its allocator are designed to be used both on the host and accelerator part of systems like the MPPA. Thus the granularity must be chosen carefully to fit both huge memory nodes (usually host) and small memory nodes (usually accelerators). For small memory nodes that do not support lazy physical page mapping, a solution to reduce overhead is to manually map parts of superpage depending on usage. This is very similar to what was done with the *Page Mapper Auto* in the version one of the allocator.

Chapter 5

Correctness

In this thesis we wanted to use formal methods to help us validate part of the code or algorithms.

At high level, we wanted to check the correctness of the *cache coherence protocol* used in Givy. We tried using a recent model checker named *Cubicle* [4] to check the model from Chapter 3. Our (mostly unsuccessful) experiments with it are discussed in Section 5.1.

At lower level, we also wanted to check the use of C11 atomics in the runtime code if possible. It is very difficult to check big systems, so we tried to model check specific data structures in Section 5.2. Some C11 atomics comments may be scattered in the implementation details in previous chapters.

5.1 Cubicle

5.1.1 Description

Cubicle [29] is a parametric model checker developed by Alain Mebsout and Sylvain Conchon. A quick description can be found on its website [4], and a more detailed explanation is available in Alain Mebsout thesis [48].

Listing 5.1 gives an example of a Cubicle model that represents a mutex used by a parametric number of threads. The model state space is defined by a set of variables defined by `var` or `array`. `var` defines a single variable, which can be a boolean, an integer, a real number, an enumeration type, or a variable in the parametric space (`proc`). `array` defines an array that is indexed by the parametric space, and can have the same types as standard variables. The model to check is then defined by 3 types of statements.

`init(a b c) predicate` defines the initial state by constraining the model variables. It is read as $\forall a, b, c\{\text{predicate}(a, b, c)\}$, with a, b, c variables picked in the parametric space. In the example, the init statement tells that both `Want` and `Crit` arrays must me set to false for all indexes. Turn is left unspecified and can be any `proc` value. The list of variables depends on the condition, it can even be empty if no value related to `proc` are constrained (this applies to all statements).

`unsafe(x y) predicate` defines an invalid state that should not be reachable from init. It is read as $\exists(x, y, z), \text{all_different}(x, y, z), \{\text{predicate}(x, y, z)\}$. Multiple unsafe

Listing 5.1: Cubicle mutex example

```
var Turn : proc (* next thread that can lock the mutex *)
array Want[proc] : bool (* tells if a thread wants to lock the mutex *)
array Crit[proc] : bool (* tells if a thread is holding the mutex *)

init (z) { Want[z] = False && Crit[z] = False }
unsafe (x y) { Crit[x] = True && Crit[y] = True }

transition req (i) requires { Want[i] = False }
{
  Want[j] := case
             | i = j : True
             | _ : Want[j]
}
transition enter (i)
requires { Want[i] = True && Crit[i] = False && Turn = i }
{
  Crit[j] := case
             | i = j : True
             | _ : Crit[j]
}
transition exit (i) requires { Crit[i] = True }
{
  Turn := . ; (* Set to random *)
  Crit[j] := case
             | i = j : False
             | _ : Crit[j] ;
  Want[j] := case
             | i = j : False
             | _ : Want[j]
}
```

statements can be used, and all will be checked by Cubicle. In the example, the only unsafe statement defines the invalid state as multiple threads holding the mutex.

Finally, transition statements take a set of parametric variables, a pre-condition, and define a state modification. Variables and arrays can be set; arrays use a pattern matching statement to tell which cells are modified.

Given this model, Cubicle must check that for all valid sequence of transitions from the initial state, the invalid state is never reached. Internally, the model checker goes backwards, and tries to reach the initial state from the invalid state by taking transitions backwards. As some variables have a parametric (unbounded) size (arrays or proc variables), the state space is infinite. Cubicle uses various methods to prune the search and reduce it to a small number of cases that can be checked exhaustively.

Some features in the syntax of Cubicle are considered extensions. They cause Cubicle to make approximations, which may prevent pruning and let it run indefinitely:

- integers and reals;

- the condition `forall p.(predicate(p))` which is true only if `predicate(p)` is true for all `p` of a parametric dimension.

Cubicle is parametric, so if it successfully checks the model, then the model is valid for all possible sizes of the parametric dimensions. So Cubicle seemed adapted to check a cache coherence protocol for a parametric number of nodes. It had already been used to validate hardware cache coherence protocol in a shared memory by the authors.

5.1.2 Modeling

We choose to check the following property: for a data-race free task graph G, a task T, a region R, T_p the closest predecessor of T in G that accessed R in `RW` mode, then T reads the value written by T_p, for all node scheduling choices. This is the property of coherence, which says that we read the value that we expected to read.

The protocol handles each region independently, so checking this property for one region is sufficient to check for all regions. The number of nodes is parametric and uses `proc` values and indexes. The next sections describe the modeling by part of the runtime.

Network

The network subsystem has been modeled with send and received buffers of size 1. The network transfer transition takes a message in a send buffer and moves it to a recv buffer atomically.

Listing 5.2: Cubicle network rule

```
transition transmit_msg (n to)
        requires {
                Init = True &&
                Sq[n] <> Msg_Empty &&
                SendTo[n] = to && Rq[to] = Msg_Empty
        }
{
        Sq[n] := Msg_Empty;
        Rq[to] := Sq[n];
        RqData[to] := SqData[n];
        RecvFrom[to] := n;
}
```

Listing 5.2 shows the message transmission rule. `Sq[proc]` and `Rq[proc]` respectively are the send and receive queue of each node (indexed by `proc` which is here a node identifier). These queues store an enum which is the protocol message type, or `Msg_Empty` for no message. `SqData[proc]` and `RqData[proc]` represents an optionally used data field that can be transmitted. It currently stores an enum of abstract values that a region can have, and this value is currently used to check the target property. `SendTo[proc]` and `RecvFrom[proc]` are respectively used to define the destination or retrieve the source node of the message. All other rules of the protocol interact with the network only by filling or reading the send / receive queues.

Protocol

The coherence protocol itself is already written as a *Labelled Transition System*, so the translation into Cubicle is straightforward. Most of the work is focused on encoding the state into Cubicle. We will take the example of the HANDLE_DATA_REQ rule. This rule handles an incoming `DataReq` message (request for a region copy from a remote node). It is defined formally as :

$$\frac{\begin{array}{l}\mathbf{dequeue}\,(\mathbf{s}[n].\,\mathbf{rq}) = \mathbf{DataReq}\,(\mathit{from}, \mathit{addr})\\ \mathbf{s}[n].\,\mathbf{r}\,[\mathit{addr}] = \mathit{reg} \wedge \mathit{reg}.\,\mathbf{own} = n\end{array}}{\begin{array}{l}\mathit{reg}.\,\mathbf{vs} := \mathit{reg}.\,\mathbf{vs} \cup \{\mathit{from}\}\\ \mathbf{enqueue}\,(\mathbf{s}[n].\,\mathbf{sq}, (\mathit{from}, \mathbf{DataAns}\,(n, \mathit{addr}, \mathit{reg}.\,\mathbf{d})))\end{array}}\ \textsc{handle_data_req}$$

Listing 5.3: One Cubicle coherence protocol rule

```
transition handle_Data_Req (n from)
        requires {
                Init = True &&
                Rq[n] = Msg_DataReq && RecvFrom[n] = from &&
                Sq[n] = Msg_Empty
        }
{
        Rq[n] := Msg_Empty;

        RegVs[from] := Valid;

        Sq[n] := Msg_DataAns;
        SqData[n] := RegData[n];
        SendTo[n] := from;
}
```

Listing 5.3 shows the Cubicle version of this rule. Its guard checks that a `DataReq` message is in the receive queue, and that the send queue is empty because we will send a reply. Note that the region is not part of the message, because we only consider one region (which is implicitly the subject of all requests). As in the formal protocol, the node then sends a `DataAns` reply back to the requester, with the up-to-date version of the region data.

Notice that the *valid set* (vs, represented by the array `RegVs[proc]`) is global and not replicated by node. In this first version of the modeling, it was chosen to simplify the ownership system to help the model checker. Thus all owner specific variables (ownership *pointer*, *valid set*, *waiting invalidation ack set*) are global, and ownership transfer is atomic. The intent was to move to the full protocol model if it managed to check the simplified one, which unfortunately was not successful.

Listing 5.4 gives a full list of network and protocol model variables. The data is currently represented by an enum type to model a data space. It has also been represented at

some point by a `proc` value which indicates the last writer to help check the target property, but it did not improve the checking. The *valid set* and *waiting invalidation ack set* are merged into `RegVS[proc]` which represents the three states by the `valid_set_state` enum. `RegRequest[proc]` is used to prevent sending the same request multiple times, as in the formal protocol.

Listing 5.4: Protocol variables

```
(* network, proc = node *)
type msg = Msg_DataReq | Msg_DataAns | Msg_OwnReq | Msg_OwnTrans |
 ↪  Msg_InvReq | Msg_InvAck | Msg_Empty
array Sq[proc] : msg
array SqData[proc] : data
array SendTo[proc] : proc
array Rq[proc] : msg
array RqData[proc] : data
array RecvFrom[proc] : proc

(* region data by node, proc = node *)
array RegValid[proc] : bool (* has valid data *)
array RegRequest[proc] : bool (* node sent a request that has not
 ↪  completed *)
type data = D_Init | D_A | D_B | D_C (* represent a set of values *)
array RegData[proc] : data (* local copy *)

var RegOwner : proc (* global, proc=owner node *)
type valid_set_state = Valid | WaitingInvAck | Invalid
array RegVs[proc] : valid_set_state (* owner-use-only, gives status by
 ↪  proc=node *)
```

Dataflow

Cubicle has already been used to check a coherence protocol based on a *threaded* execution model. A threaded model has N threads that can emit any memory access requests in any order, which makes modeling the accesses simple (not much structure to represent). Instead, as Givy relies on a dataflow task graph model without data-race, its constrained structure must be represented in Cubicle. Three different strategies have been tried, but none was successful enough to let the model checker check the property.

Fixed Graph Modeling: Init State The first strategy tried encodes the data-flow graph in the parametric dimensions (using `array[proc]`). As a `proc` value can now represent either a task or a node, the `Sort[proc]` array is using to *type* `proc` values and keep the node and task parametric dimensions independent. The task status is encoded using another set of array variables (`TaskState`, `TaskAccessMode`, `TaskAccessData`, `TaskSC`, `TaskDep`). Task manipulation transitions (which manage task state, and when task trigger protocol requests) are then expressed naturally in terms of these variables.

Listing 5.5: Fixed graph modeling: init state

```
type sort = Node | Task
array Sort[proc] : sort
transition setup_example (ta tb ra)
        requires { Init = False && Sort[ta] = Task && Sort[tb] = Task &&
         ↪  Sort[ra] = Node }
{
        (* A[W] -> B[R] example *)
        TaskState[id] := case
                | id = ta : TS_Created
                | id = tb : TS_Created
                | _ : TaskState[id];
        TaskAccessMode[id] := case
                | id = ta : RW
                | id = tb : RO
                | _ : TaskAccessMode[id];
        TaskAccessData[id] := case
                | id = ta : D_A
                | id = tb : D_A
                | _ : TaskAccessData[id];
        TaskSC[id] := case
                | id = ta : 0
                | id = tb : 1
                | _ : TaskSC[id];
        TaskDep[ta, tb] := True;
        (* set ra as initial region owner *)
        RegOwner := ra;
        RegValid[ra] := True;
        Init := True;
}
```

The first step was to try to encode a fixed task setup into the model. This requires to have an init state where a fixed set of `proc` values chosen to represent the fixed chosen task graph. The `init()` statement is not adapted, as it constrains `proc` values by a forall statement, whereas we want to *pick* unique `proc` values in the parametric space and configure arrays with them. Thus we use the *init trick*, where a *setup transition* is forced to be taken before every other transition. A global boolean value `Init` is initialized as false, and required to be true by every model transition except for the setup transition which sets its to true. Listing 5.5 shows an example of a setup transition for a fixed graph with two tasks. Transitions *pick* `proc` values with an *exists*, and they are automatically all different from each other, which is exactly what we wanted.

A few protocol rules use forall statements, like TASK_STARTS_RUNNING, and TASK_BECOMES_ELIGIBLE. TASK_STARTS_RUNNING does a forall on regions, which becomes a single check in this modeling as we only consider one region independently. However, TASK_BECOMES_ELIGIBLE must check that all dependencies have been cleared, which requires a reduction mechanism even in our modeling

as we may have multiple tasks. Both available mechanisms – using an integer synchronization counter, or an array of booleans with a forall check – are extensions to Cubicle. Cubicle indicates that the model checker may not converge if these are used, as the model checker must do approximations that are outside the theory. In the end, this strategy did not converge on a proof, even for very simple graphs like the one in Listing 5.5 (2 tasks).

Fixed Graph Modeling: CTC The second approach was to try to remove the use of Cubicle extensions which may prevent checking. Even if the set of dependencies of a task is dynamic, for each task in a fixed task graph it is a fixed set. Thus the idea was to duplicate variables and rules for each task. That way, for each task, the TASK_BECOMES_ELIGIBLE rule of that task only has to check a fixed set of dependencies, which removes the use of *forall*.

Listing 5.6: Fixed graph modeling: CTC rule

```
transition @Tasks @ t@0@_becomes_eligible_starts_caching (node)
        requires {
                Task_@0@_State = TS_Created && Task_@0@_Node = node &&
                @0.dep@ (&& Task_@1@_State = TS_Finished)
        }
{
        Task_@0@_State := TS_Caching;
}
```

Generating the task specialized rules was done with a small custom python tool named *Cubicle Template Compiler* (available at github[1]). An example of the CTC version of TASK_BECOMES_ELIGIBLE is presented in Listing 5.6. CTC adds annotations based on `@` to the Cubicle grammar. Grammar elements tagged with these annotations will be replicated according to an external data set using a JSON format (example in Listing 5.7). Elements which can be annotated are : transitions, unsafe statements, part of the predicate (*or* and *and* statements), enum declarations, variables declarations and assignements.

The Listing 5.6 example reads as follows : for each task in the fixed graph, generate a task specific "becomes eligible" rule. The rule name includes the task name with `@0@`, to differentiate generated rules. The transition guard then references the task specific variables (there is no `proc` array anymore, tasks are not part of the parametric space). The `@0.dep@ (&& Task_@1@_State = TS_Finished)` statement will generate a fixed conjunction of checks for each dependency task, which replaces the forall statement.

The external graph specification format would have allowed a fast generation of test cases, even compared to the previous version of the modeling. However, it still was not able to check the property, except for few trivial graphs (with no writes). A possible reason is the increased search space due to replicated rules.

Building Step Modeling The two previous attempts focused on encoding a specific task graph instance in Cubicle structures. As the protocol relies on the task graph being

[1] `https://github.com/lereldarion/ctc`

Listing 5.7: CTC json graph specification

```
{
    "Regions": [ 0 ],
    "Tasks": {
        "A": {
            "accesses": {
                "0": { "mode": "C" }
            },
            "dep": []
        },
        "B": {
            "accesses": {
                "0": { "read": "D_A", "mode": "RW" }
            },
            "dep": [ "A" ]
        },
        "C": {
            "accesses": {
                "0": { "read": "D_B", "mode": "RO" }
            },
            "dep": [ "B" ]
        }
    }
}
```

data-race-free, it is likely that Cubicle needs to use this information in its proof. However, with a given graph instance, Cubicle would have to discover that it is data-race-free on its own, without knowledge of how the graph instance was built.

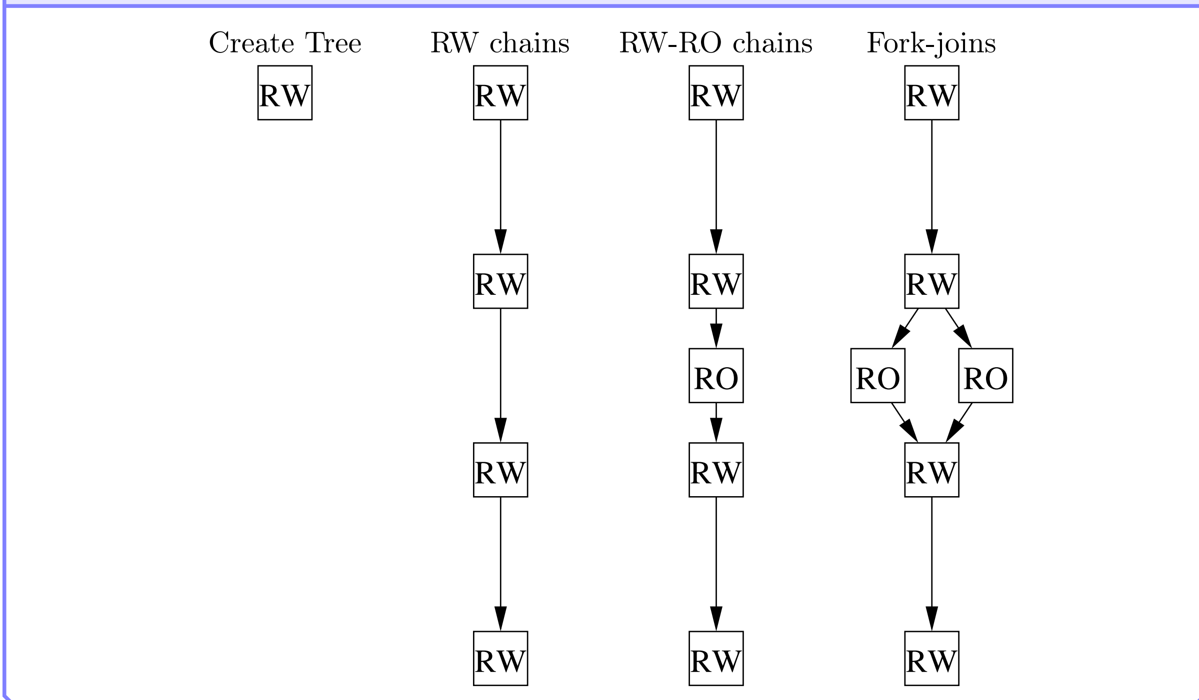
A way to help Cubicle would be to provide him with useful properties to check, like the data-race-free property. Some model checkers allow users to give them intermediate steps as a guide to accelerate the proof. Cubicle has an `invariant(){...}` statement in its parsing grammar, but unfortunately it is not documented, and seems unfinished.

Another idea was to include the task graph construction in the Cubicle modeling step. This seemed complex at first, which is why user-provided graphs were tried first. If it worked, it would allow to check the protocol with entire graph families instead of single task graphs. The set of verified graphs would be the set of graph that can be generated by the construction process. This set would depend on the complexity of the construction process, but is not guaranteed to include all data-race-free graphs.

A simple construction process was tried at first, illustrated in Fig. 5.1. It builds a graph in multiple steps, with each step enabling some constructions rules that can be repeatably used to build many graphs. The first step is to create the single root RW task. Then the graph is expanded to a linear chain of RW tasks. After that, RO tasks are inserted in the chain. The last encoded step was to add RO tasks in parallel with existing RO tasks, creating fork-join patterns.

The graph construction process was tested independently from the protocol. It was

Figure 5.1: Cubicle task graph construction steps.



tested by checking that all graph generated are indeed *data-race-free*. The Cubicle encoding of DRF properties is shown in Listing 5.8. Unfortunately, Cubicle was not able to check that the generated graph was still *data-race-free* after introducing the RO tasks (RW-RO chains step). As the data-race-free properties are simpler than the protocol, full verification of the protocol with the graph construction process was not tested.

5.1.3 Conclusion

Cubicle was not sufficient to test the cache coherence protocol of Givy. It has already been used by the authors to check one hardware shared-memory coherence protocol successfully, which motivated our experiments for Givy's protocol. However most hardware-level coherence protocols do no depend on a specific memory access pattern like Givy's protocol do (data-race-free task graph). This make their encoding in Cubicle easier, and more likely to check quickly if the protocol is valid. This can be seen in our experiments, where encoding the task-graph was the difficult part, requiring Cubicle extensions or tricks like CTC.

Verification of the protocol could be done by hand. However one of the goal in Givy was to be able to iterate on the protocol design, so writing many proofs manually is not very practical in this case. The Givy protocol could be encoded in other parametric model checkers, like Ivy [58]. A good strategy would be to test the graph construction process to check if the model checker is able to verify graph properties.

Listing 5.8: Graph construction: DRF properties

```
(* Access[task, region] = RW | RO | NU (not used) *)
(* DepTrans[taskA, taskB] = bool : transitive dependency from A to B *)
unsafe (ta tb r) {
        Step = Check && ta <> tb &&
        Sort[ta] = Task && Sort[tb] = Task && Sort[r] = Region &&
        Access[ta, r] = RW && Access[tb, r] = RO &&
        DepTrans[ta, tb] = False && DepTrans[tb, ta] = False
} (* RO parallel to RW *)
unsafe (ta tb r) {
        Step = Check && ta <> tb &&
        Sort[ta] = Task && Sort[tb] = Task && Sort[r] = Region &&
        Access[ta, r] = RW && Access[tb, r] = RW &&
        DepTrans[ta, tb] = False && DepTrans[tb, ta] = False
} (* RW parallel to RW *)
unsafe (ta tb) {
        Step = Check && ta <> tb &&
        Sort[ta] = Task && Sort[tb] = Task &&
        DepTrans[ta, tb] = True && DepTrans[tb, ta] = True
} (* cyclic deps *)
```

5.2 Shared Memory Data Structures

Each node instance of the runtime is designed to work on a *shared memory architecture*. The runtime code – allocator, coherence, scheduler – contains a lot of parallel accesses to shared memory structures. *Data races* are any access to a shared memory object in parallel with a write access to the same object. They are almost always present in non trivial parallel code. A data race results in a non deterministic program state, which if not properly handled leads to *undefined behavior* and program crash. However a few carefully controlled data races are commonly used: *mutexes* (race on the `locked` flag), *synchronization barriers*, and others parallel control flow primitives. These primitives work by hiding the non-deterministic behavior in a meaningful API : for example, `mutex_lock()` hides the flag access (data race) and creates a defined behavior of « blocks executions until the mutex is locked ».

These *useful* data races are difficult to implement correctly, because they rely on a specific pattern of accesses between threads. Both hardware and compiler have many mechanisms designed to improve single threaded performance : hardware caches, instruction reordering in superscalar processors, instruction reordering by the compiler, and other compiler optimizations. All of these mechanisms are fine in a single thread point of view but will change the pattern of memory accesses, which may break the properties on which the useful data race relies.

To allow implementing these data races, both processors and compiler provide some kind of *annotations* that can be used to mark parallel accesses. It is then expected that the processor / compiler will recognize the annotations and prevent some reordering of these special memory accesses, thus preserving the desired access pattern. At the processor

level, this is done by providing special instructions like *memory barriers*, or *read-modify-write* atomic instructions. Memory barriers will flush the cache and prevent reordering of memory load and store instructions in superscalar processors. Read-modify-write instructions, like *Compare-And-Swap* (CAS), will read, modify, and rewrite a value in memory atomically. At the compiler level – in C / C++ – there were no dedicated mechanisms for a long time: programmers used the `volatile` keyword, or *compiler builtin functions* which mapped to the processor special instructions (not guaranteed to maintain access ordering). Since the C11 / C++11 standard, the compiler provides *atomic* variables and memory access primitives which generates the processor special instructions, and prevent reordering [1].

In practice, designing these *useful data races* is still hard. Programmers need to take into account every possible access pattern and check that they maintain the API-defined behavior. Ongoing work has been done to formalize the behavior of data race memory accesses; a formalization is called a *memory model*. Work initially focused at the processor level, for example with the thesis of Jade Alglave [12]. More recent work now aims to formalize the C11 / C++11 memory model [19]. Memory model formalization is a stepping stone that allows verification of properties in programs with data races. Several model checkers have been developed [3, 54], which can be very helpful to help programmers. Compiler optimizations can be formally proved to preserve the data race access patterns [49].

In Givy, shared memory structures have been developed using the C11 / C++11 memory model to allow portability. No formal proof has been made, however small localized data races have been checked with the CPPMEM model checker [3].

5.2.1 C11 / C++11 Memory Model

This section briefly gives a non formal overview of the C11 memory model.

Since C11 / C++11 , a data race involving *normal* memory accesses has *undefined behavior*. This allows the compiler to use many optimizations by assuming that the code is sequential (no parallel threads). To prevent undefined behavior, variables which racy accesses must be declared as `atomic`. In C++11, this is done by using the `atomic<T>` template type instead of `T`. In C11, this is done by using the `_Atomic` type specifier, or one of the many type aliases for basic types like `atomic_int`.

> In theory `atomic<T>` / `_Atomic` can be applied to any type `T`. In practice, most processors only provide access atomicity and special instructions (like *Compare-And-Swap*) for small data sizes. Bigger types must emulate atomicity and special instructions, which can have low performance. For example, GCC and Clang will implement accesses as calls to the `libatomic` emulation library. This library is surprisingly not linked automatically, resulting in a linker error if `atomic<T>` with a too large `T` type is used in the program. Thus in my view it is better to use only small basic types; standard macros provide information about which types are natively supported by `atomic<T>`.

Memory accesses must be tagged by an *access mode* that provide different semantics. *Relaxed* accesses provide the *weakest* semantics (least amount of guarantees). Relaxed

accesses should not be deleted (a valid sequential optimization is removal of redundant stores). Reordering are still allowed as long as they do not break data flow dependencies in each individual thread. Thus the memory access pattern, as viewed by other threads, is not guaranteed to be conserved.

Release and *acquire* accesses prevent some compiler / processors reorderings around them. A *load acquire* prevents subsequent loads to happen before itself. A *store release* prevents previous stores to happen after itself. A load acquire returning a value set by a store release ensures that the acquire thread gets access to all previous stores from the release thread. All types of accesses (even non atomic) are ordered. Mutexes will typically implement `lock()` with at least acquire ordering, and `unlock()` with release ordering. This ensures that accesses inside the critical section (even non atomic ones) get values from the previous sections, and modifications will be propagated to the next sections.

Sequentially consistent (SC) accesses are the strongest accesses. All sequentially consistent accesses from all threads are totally ordered as if those accesses were interleaved in a global sequence. A sequentially consistent access is also release and acquire.

Stronger access types will generally have more runtime cost : flush the cache, prevent optimizations. However some architectures have strong properties on their accesses. For example x86 basic load and store instructions have release-acquire semantics, making them no more expensive than relaxed accesses. At the other end of the spectrum, the MPPA provides even weaker semantics than relaxed, and have also have instructions that can bypass the cache entirely.

Memory barriers are instructions that are not memory accesses, but provide the reordering constraints around them. They exist in release, acquire and sequentially consistent flavors. *Compare-and-swap* is also available.

5.2.2 Cuckoo Hash Table

In this section we describe the use of C11 atomics in the hash table used in the *page mapper fixed* of the V1 allocator (see Section 4.4.3). This is a *concurrent cuckoo hash table* adapted from Memc3 [36]. This new implementation use C11 atomics instead of `volatile` variables, and has been checked using the CPPMEM model checker [3]. The following description explains the reasoning behind the use of atomics, but is not a formal proof.

The hash table data structures are described in Listing 5.9. They implement the structures described in Fig. 4.4. The hash table – which is actually a hash set – stores the keys in buckets. The array of buckets is not resizable, thus the `buckets` pointer is does not change and does not need to be atomic. Bucket key values are atomic because they will generate data races, as the table is lock-free. Synchronization counters are used to indicate changes to the table to readers, and generate data-race too. Hash functions associate two buckets and one counter to each key. The synchronization counter of a key is incremented two times on each modification: before and after the modification. Counters start at 0, thus odd counters indicate an ongoing modification, while even counters indicate that readers can do read only operations. If a counter changed during the read operation, it must be restarted as the key status might have been changed.

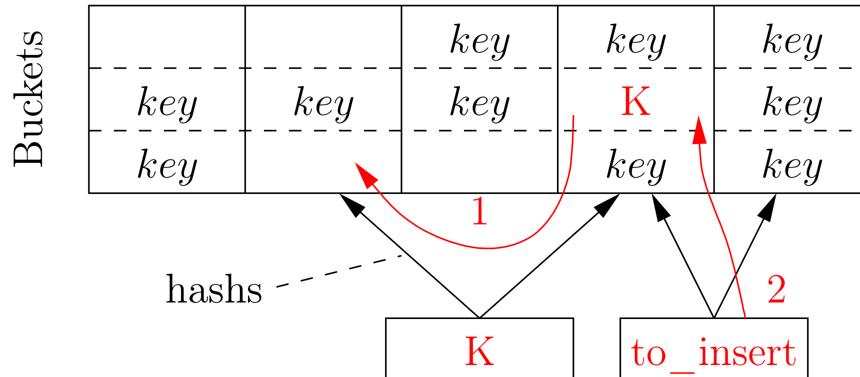
This implementation assumes that changes (insertions and deletions) are serialized,

Listing 5.9: Cuckoo hash-table: structures

```
struct hash_bucket_t {
  atomic_uintptr_t values[PMF_HASH_BUCKET_SIZE];
};
struct hash_t {
  // Array of buckets
  size_t nb_buckets;
  struct hash_bucket_t * buckets;
  // Array of synchronization counters
  atomic_uint_least32_t sync_counters[PMF_HASH_LOCK_ARRAY_SIZE];
};
```

so no modification / modification data race can occur. The only read operation is `contains(table, ptr)` which tests if `ptr` is in `table`. Its code is given in Listing 5.10. Shared data is only accessed in read-only, so parallel calls of `contains()` do not generate any data race. Thus, the only case to look at is a modification in parallel with a call to `contains()`.

Figure 5.2: Example of cuckoo move sequence for insertion. The `to_insert` key is to be inserted, but both buckets allowed by hash functions are full. the `K` key is chosen in one bucket, and moved in an empty space in its other allowed bucket. It frees a space for `to_insert` which is then inserted.



Both modification operations use the `move_key()` primitive (in Listing 5.11) to perform all atomic changes. `move_key()` moves a key value from one bucket cell to another, while updating the key synchronization counter. The origin bucket cell is set to `NULL`. For the sake of this explanation, a deletion can be considered as a move towards a *dummy bucket*. An insertion is done in two phases. The table is first traversed to find a place to insert the new key, or the sequence of key moves to free the required place. Then the sequence of key moves is performed using `move_key()`, with the final insertion being a move from a *dummy bucket*. Fig. 5.2 illustrates a small sequence of moves for key insertion.

Thus the data races of this hash table implementation are only `move_key()` (List-

Listing 5.10: Cuckoo hash-table: contains

```
int contains (struct hash_t * table, const void * ptr) {
  uint32_t h[2], hl;
  uintptr_t ptr_value = (uintptr_t) ptr;
  make_hashs (table->nb_buckets, ptr_value, h, &hl);
  while (1) {
    // Wait for even counter
    uint_least32_t counter_snapshot = atomic_load_explicit
     ↪  (&table->sync_counters[hl], memory_order_acquire);
    if (! (counter_snapshot & 0x1)) {
      // Read buckets to find ptr
      int found = 0;
      for (uint32_t i = 0; i < 2 && !found; ++i) {
        struct hash_bucket_t * bucket = &table->buckets[h[i]];
        for (uint32_t j = 0; j < HASH_BUCKET_SIZE && !found; ++j)
          found = atomic_load_explicit (&bucket->values[j],
           ↪  memory_order_relaxed) == ptr_value;
      }
      atomic_thread_fence (memory_order_acquire);

      // Check counter is still the same before giving result
      if (atomic_load_explicit (&table->sync_counters[hl],
       ↪  memory_order_relaxed) == counter_snapshot)
        return found;
    }
  }
}
```

ing 5.11) in parallel with a `contains()` (Listing 5.10). The invalid outcome of this data race is `contains()` returning an answer based on an intermediate state of the `move_key()`. For example, during an intermediate move of a different key from the one being inserted, it is invalid for `contains()` to indicate that this moved key is not in the table.

Contains design `contains()` reads both buckets associated to a key, in order to check its presence or absence. The read of bucket values is enclosed by two loads of the associated synchronization counter. If the counter changed during the check, the `contains()` operation is restarted. The initial synchronization counter load is *acquire* to ensure that we get the last modifications associated to the key: modifications are supposed to have changed the counter with store release. All bucket loads are relaxed for performance. They are followed by a acquire barrier before the second counter load (which is relaxed). Finally, if the counter snapshots match, the result of the bucket check is returned.

Move_key design `move_key()` performs the bucket value changes, enclosed by two increments of the synchronization counter. The first increment is done using relaxed

Listing 5.11: Cuckoo hash-table: move

```
void move_key (atomic_uint_least32_t * counter_ptr, atomic_uintptr_t *
 ↪  from, atomic_uintptr_t * to) {
  // Get initial value (even)
  uint_least32_t counter = atomic_load_explicit (counter_ptr,
   ↪  memory_order_relaxed);
  // Add 1 to indicate ongoing modification, and fence
  atomic_store_explicit (counter_ptr, counter + 1, memory_order_relaxed);
  atomic_thread_fence (memory_order_release);
  // Move
  uintptr_t val = atomic_load_explicit (from, memory_order_relaxed);
  atomic_store_explicit (from, (uintptr_t) NULL, memory_order_relaxed);
  atomic_store_explicit (to, val, memory_order_relaxed);
  // Add 1 to indicate modification finished
  atomic_store_explicit (counter_ptr, counter + 2, memory_order_release);
}
```

accesses, then followed by a release barrier, and then the relaxed bucket modifications. The release barrier works in pair with the acquire barrier of `contains()`. If any bucket load in `contains()` reads a value from one of the bucket stores of `move_key()`, then the two barriers ensures that the first synchronization counter increment will be visible after the acquire barrier in `contains()`. Thus, if `contains()` reads bucket values which are being modified, it is guaranteed to fail on the synchronization check and start again. After all modifications, the counter is incremented again with a release store to re-enable read operations.

Conclusion The hash table only uses release and acquire accesses at most. Release-acquire accesses are cheap on x86 architectures, which is good for performance.

`move_key()` and `contains()` have been checked in the CPPMEM [3] model checker. CPPMEM only supports small code snippets in parallel, and returns all possible outcomes that are allowed by the formal memory model. `contains()` was written as a single try (no loop). In the setup, a key `K` was being moved, while a `contains()` operation tested its presence. Expected outcomes were either a successful `contains()` test detecting the presence of `K`, or a failed `contains()` which must be retried. A bad outcome would be a successful `contains()` indicating that `K` is absent. All outcomes computed by CPPMEM were in the expected set, thus proving the correctness of the hash table implementation.

Chapter 6

Conclusion

Motivations We initially aimed to design and evaluate a DSM runtime to efficiently execute irregular parallel applications on MPSoC architectures. The MPPA (from the Kalray company) was our initial target: it is a MPSoC with a network linking 16 nodes of 16 cores. The runtime was to use tasks with dependencies, dynamic scheduling, and use a software coherence protocol as no coherence is provided on the MPPA. The *Owner Writable Memory* concept was selected for the coherence protocol. It allows metadata (owner node) to move along with data, so moving computation should also move metadata, reducing costly remote accesses to metadata.

Another motivation behind OWM was to try to add mutable objects references to a pure data-flow task model, and see how much complexity is needed to implement it in a distributed runtime system.

Timeline The initial step was to formally define and refine the Owner Writable Memory concept, based on a prototype implementation by Boris Arnoux and discussions with my advisors. This lead to the formal specification presented in Chapter 3.

The next step was to create a new runtime with the lessons from the formalization, called Givy. We chose to create a global address space DSM using *real pointers* as it should help port existing code (no modification of pointer-based structures). The need of a carefully designed allocator subsystem was identified: the allocator must be GAS-aware, avoid collision in the GAS, and have the best possible performance.

A first implementation (version 1) of the allocator subsystem was completed and its design and evaluation was published at ISMM [39]. It demonstrated that a GAS allocator design with almost no network traffic and reasonable performance was possible. A second implementation (version 2) was proposed, with an incrementally improved design addressing flexibility and performance deficiencies of the version 1 allocator. This implementation also shifted development towards a x86 prototype of the Givy runtime (instead of the MPPA, which slowed development due to work in progress on system APIs). The new design better supports huge memory footprints that can be found outside embedded architectures. It also integrates much deeper with the coherence protocol, managing metadata placement and providing critical low level primitives for allocation but also metadata access. Both implementations are described in Chapter 4.

Finally, development of the network layer and coherence procotol of Givy started. Due to time constraints, it has not been completed by the end of the thesis. The overall design

of the runtime has been described in Chapter 2.

In parallel with runtime implementation, we attempted to formally and mechanically verify the main properties of the protocol, using the Cubicle model checker. These attempts were only partially successful; they are detailed in Chapter 5.

Contributions

- Overall design of a the Givy GAS runtime, which is task based, is a library approach, and uses real pointers as GAS references.
- Formalization of a software cache coherence protocol for the Givy GAS runtime, based on the Owner Writable Memory concept.
- Translation of the coherence protocol in a the language of the Cubicle model checker, and description of validation attempts.
- Implementation of two versions of the allocator subsystem of the Givy GAS runtime: the prototype version 1, and an improved stable version 2.

Perspectives In Givy and Myrmics [45], which are both real-pointer GAS runtimes, the allocator subsystem has been critical to managing the virtual address space of each cluster node. Using real pointers requires the runtime to manage coherence metadata itself (creation, placement). Both coherence metadata and memory allocation have similar properties, so a key takeaway of Givy is that integrating coherence metadata in the allocator greatly helps with runtime design. This is reflected in the transition from the version 1 to version 2 of the allocator. On embedded architectures, integration could go even further, by merging the virtual memory system (page tables) in the allocator itself.

However choosing real pointers adds a lot of complexity to the design compared to fat pointers. Moreover, the argument of keeping pointer-based data structures unmodified from sequential code is weak. To efficiently parallelize, the computation must be split, especially in Givy where tasks boundaries are the only place for data transfers. Thus any application ported to Givy must be heavily modified, either manually or with a compiler pass, which could also introduce fat pointers in the process.

The region system of Myrmics is interesting and simplifies specification of sets of objects. However it is limited by the tree hierarchy (a region can only be owned by one region); a directly acyclic graph model would be more flexible. In Givy, the splitting model is effective for linear algebra, but less for pointer based structures. The region set is very unpractical in my opinion, and specific cases for specific data structures (trees, lists) would need too much runtime specialization. A better way would be for applications programmers to explain to the runtime how to interpret a pointer-based structure. It could be manual (specific functions overloads for user types in C++), or semi-automatic if the language has reflection capabilities (C++{in a long time}).

Real pointers also force where remote data copies must be placed in the local node's virtual memory. Thus for every allocator placement scheme, it is possible to build a task scenario which will generate a very inefficient usage of memory. An extremely fragmented example would be a task that requires many small blocks (of a few bytes each), all owned

by remote nodes, and none on the same virtual memory page. Fat pointers have the flexibility to place remote data copies in any place, and can pack them appropriately.

The Givy runtime, in its current design, is difficult to use. Moreover, due to its very low level interface, it is difficult to ensure that the DRF property is valid in Givy programs. Detecting violations at runtime would require adding more metadata to track the system state more precisely, which would increase the runtime overhead. Givy could be used as a target for higher level compiler, which would generate runtime API calls, provide necessary annotations, and enforce properties at high level. Many other DSM already use a compiler to generate runtime calls, like UPC [26], Myrmics [45], Legion [20]. Another option would be to integrate it in a software framework like STAPL [16].

This thesis initially included a compiler part, which was not done due to a greater focus on runtime and memory allocation. The idea was to go beyond simple runtime API call generation, and optimize at the task graph level. Possible optimizations would be: generation of target specific task code alternatives for heterogeneous targets, automatic coalescing of fine grained tasks if inefficient on the target architecture, compiler controlled division of work that adapts to targets. This dual compiler and runtime approach is in fact the focus of the CORSE research team, where this thesis was done.

One of my personal motivations for this thesis was the informal *Grmbl language* project. The idea is to create a language that is inherently data driven, asynchronous, represented as data flow. It would natively expose parallelism and data flow that could be exploited by DSM like systems. While this is for now an early project with a blurry definition and a scope which is probably too wide, progress is being made on the programming language front. Memory management semantics are already making their way in programming languages: *smart pointers* of C++11, *ownership* and *borrow* semantics in Rust (which are statically checked). Type systems are also expanded to encode and check more complex properties (*Mezzo* language [62]). Thus language and compilers might soon be able to express and manipulate various memory semantics. Compilers able to manipulate and translate memory semantics related code would greatly help checking and compiling code for DSM systems like Givy, if program data could be adapted to fit within the runtime supported memory management structures.

List of Figures

1.1 Trends of processors 1
1.2 Energy estimations of US datacenters 2
1.3 Distributed and Shared memory 4
1.4 Distributed and Shared memory properties 5
1.5 Structure of an SGI UV-2000 NUMA machine 11
1.6 PGAS vs SM vs DM 12
1.7 Givy task graph example: blocked sparse matrix multiplication $C = A \star B$ 15

2.1 Example of a virtual memory layout 18
2.2 Main components of Givy 21
2.3 Structure of a task frame 22
2.4 Life of task (possible states) 23
2.5 Execution model example program 25
2.6 Execution model example trace 26
2.7 MPPA architecture 27
2.8 Region split / merge 30
2.9 Stanford legion runtime stack 33
2.10 Grappa delegate example 34

3.1 Example program and execution trace. 39
3.2 Models architecture. 41
3.3 VI protocol transition system. 42
3.4 MSI protocol transition system. 43

4.1 GAS layout example 59
4.2 GAS layout with node areas 60
4.3 Givy overview with v1 allocator 63
4.4 Cuckoo hash table structure 65
4.5 Page Mapper Auto structure 67
4.6 Local Allocator structure 69
4.7 V1 benchmarks: execution time 72
4.8 V1 OpenStream benchmarks 73
4.9 V1 Givy benchmarks: memory footprint 74
4.10 V1 Hoard benchmarks: memory footprint 75
4.11 V1 benchmarks: proxy allocator 76
4.12 V2 allocator internal structure 78

5.1 Cubicle task graph construction steps. 89
5.2 Cuckoo move sequence for insertion 93

List of Listings

1.1 Listing 1.1: Data transfer in distributed memory 6
1.2 Listing 1.2: Data transfer in shared memory (naive example) 7
1.3 Listing 1.3: Example of UPC constructs 12
1.4 Listing 1.4: Givy example: blocked sparse matrix multiplication $C = A \star B$ 14
2.1 Listing 2.1: Interesting C++ constructions 28
4.1 Listing 4.1: Dynamic memory allocation API in C 54
4.2 Listing 4.2: Naive implementation of `posix_memalign()` 55
4.3 Listing 4.3: Allocator V1 allocation primitives 62
5.1 Listing 5.1: Cubicle mutex example 82
5.2 Listing 5.2: Cubicle network rule 83
5.3 Listing 5.3: One Cubicle coherence protocol rule 84
5.4 Listing 5.4: Protocol variables 85
5.5 Listing 5.5: Fixed graph modeling: init state 86
5.6 Listing 5.6: Fixed graph modeling: CTC rule 87
5.7 Listing 5.7: CTC json graph specification 88
5.8 Listing 5.8: Graph construction: DRF properties 90
5.9 Listing 5.9: Cuckoo hash-table: structures 93
5.10 Listing 5.10: Cuckoo hash-table: contains 94
5.11 Listing 5.11: Cuckoo hash-table: move 95

Bibliography

[1] C++11 atomics api. http://en.cppreference.com/w/cpp/atomic. Accessed: 18/07/2017.

[2] Corba standard. http://www.omg.org/spec/CORBA/Current/. Accessed: 17/10/2016.

[3] Cppmem. http://svr-pes20-cppmem.cl.cam.ac.uk/cppmem/. Accessed: 24/08/2016.

[4] Cubicle website. http://cubicle.lri.fr/. Accessed: 17/10/2016.

[5] Intel xeon phi. https://software.intel.com/sites/default/files/Intel%C2%AE_Xeon_Phi%E2%84%A2_Coprocessor_Architecture_Overview.pdf. Accessed: 15/10/2016.

[6] Kalray. http://www.kalrayinc.com. Accessed: 10/09/2016.

[7] Sgi mpi implementation. https://www.sgi.com/pdfs/4236.pdf. Accessed: 17/10/2016.

[8] Stapl website. https://parasol.tamu.edu/stapl/. Accessed: 10/04/2018.

[9] Tcmalloc. http://gperftools.googlecode.com/svn/trunk/doc/tcmalloc.html. Accessed: 24/08/2016.

[10] Tilera. http://www.tilera.com. Accessed: 10/09/2016.

[11] Martin Aigner, Christoph M Kirsch, Michael Lippautz, and Ana Sokolova. Fast, multicore-scalable, low-fragmentation memory allocation through large virtual memory and global data structures. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 451–469. ACM, 2015.

[12] Jade Alglave. A shared memory poetics. *These de doctorat, L'université Paris Denis Diderot*, 2010.

[13] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in weak memory models. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*. Springer, 2010.

[14] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Litmus: running tests against hardware. In *Proceedings of the 17th international conference on Tools and algorithms for the construction and analysis of systems: part of the joint European conferences on theory and practice of software*, TACAS'11/ETAPS'11, pages 41–44, Berlin, Heidelberg, 2011. Springer-Verlag.

[15] Cristiana Amza, Alan L Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *Computer*, 29(2):18–28, 1996.

[16] Ping An, Alin Jula, Silvius Rus, Steven Saunders, Tim Smith, Gabriel Tanase, Nathan Thomas, Nancy Amato, and Lawrence Rauchwerger. Stapl: An adaptive, generic parallel c++ library. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 193–208. Springer, 2001.

[17] Boris Arnoux. Tstar runtime. http://github.com/borisarnoux/TStar, 2012.

[18] Scott Atchley, David Dillow, Galen Shipman, Patrick Geoffray, Jeffrey M. Squyres, George Bosilca, and Ronald Minnich. The common communication interface (CCI). In *19th Annual IEEE Symposium on High-Performance Interconnects*, August 2011.

[19] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing c++ concurrency. In *ACM SIGPLAN Notices*, volume 46, pages 55–66. ACM, 2011.

[20] Michael Edward Bauer. *Legion: programming distributed heterogeneous architectures with logical regions.* PhD thesis, STANFORD UNIVERSITY, 2014.

[21] Emery D Berger, Kathryn S McKinley, Robert D Blumofe, and Paul R Wilson. Hoard: A scalable memory allocator for multithreaded applications. *ACM Sigplan Notices*, 35(11):117–128, 2000.

[22] Emery D Berger, Benjamin G Zorn, and Kathryn S McKinley. Composing high-performance memory allocators. In *ACM SIGPLAN Notices*, volume 36, pages 114–124. ACM, 2001.

[23] Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[24] Dan Bonachea. Gasnet specification, v1. 1. 2002.

[25] Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, and Sagnak Taşirlar. Concurrent collections. *Sci. Program.*, 18:203—217, 2010.

[26] William W Carlson, Jesse M Draper, David E Culler, Kathy Yelick, Eugene Brooks, and Karen Warren. *Introduction to UPC and language specification.* Center for Computing Sciences, Institute for Defense Analyses, 1999.

[27] Bradford L Chamberlain, David Callahan, and Hans P Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[28] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *Acm Sigplan Notices*, 40(10):519–538, 2005.

[29] Sylvain Conchon, Amit Goel, Sava Krstić, Alain Mebsout, and Fatiha Zaïdi. Cubicle: A parallel smt-based model checker for parameterized systems. In *International Conference on Computer Aided Verification*, pages 718–724. Springer, 2012.

[30] David Culler, J. P. Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann, 1 edition, August 1998.

[31] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[32] Benoît Dupont de Dinechin, Renaud Ayrignac, Pierre-Edouard Beaucamps, Patrice Couvert, Benoit Ganne, Pierre Guironnet de Massas, François Jacquet, Samuel Jones, Nicolas Morey Chaisemartin, Frédéric Riss, et al. A clustered manycore processor architecture for embedded and accelerated applications. In *HPEC*, pages 1–6, 2013.

[33] Benoît Dupont de Dinechin, Pierre Guironnet de Massas, Guillaume Lager, Clément Léger, Benjamin Orgogozo, Jérôme Reybert, and Thierry Strudel. A distributed run-time environment for the kalray mppa®-256 integrated manycore processor. *Procedia Computer Science*, 18:1654–1663, 2013.

[34] Mattias De Wael, Stefan Marr, Bruno De Fraine, Tom Van Cutsem, and Wolfgang De Meuter. Partitioned global address space languages. *ACM Computing Surveys (CSUR)*, 47(4):62, 2015.

[35] Daniel Drake and Johannes Berg. Unaligned memory access, linux kernel documentation. `https://www.kernel.org/doc/Documentation/unaligned-memory-access.txt`. Accessed: 20/08/2016.

[36] Bin Fan, David G Andersen, and Michael Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *NSDI*, pages 371–384, 2013.

[37] Message Passing Forum. Mpi: A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994.

[38] Balazs Gerofi, Masamichi Takagi, and Yutaka Ishikawa. Exploiting hidden non-uniformity of uniform memory access on manycore cpus. In *Euro-Par 2014: Parallel Processing Workshops*, pages 242–253. Springer, 2014.

[39] François Gindraud, Fabrice Rastello, Albert Cohen, and François Broquedis. A bounded memory allocator for software-defined global address spaces. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management, Santa Barbara, CA, USA, June 14 - 14, 2016*, pages 78–88, 2016.

[40] Bruce L Jacob and Trevor N Mudge. A look at several memory management units, tlb-refill mechanisms, and page table organizations. In *ACM SIGOPS Operating Systems Review*, volume 32, pages 295–306. ACM, 1998.

[41] Abhishek Kulkarni and Andrew Lumsdaine. Active global address space (agas): Global virtual memory for dynamic asynchronous many-tasking (amt) runtimes.

[42] Doug Lea. dlmalloc. `http://g.oswego.edu/dl/html/malloc.html`. Accessed: 20/08/2016.

[43] Ran Liu and Haibo Chen. Ssmalloc: a low-latency, locality-conscious memory allocator with stable performance scalability. In *Proceedings of the Asia-Pacific Workshop on Systems*, page 15. ACM, 2012.

[44] Spyros Lyberis. *Myrmics: A scalable runtime system for global address spaces*. PhD thesis, PhD thesis, University of Crete, 2013.

[45] Spyros Lyberis, Polyvios Pratikakis, Iakovos Mavroidis, and Dimitrios S Nikolopoulos. Myrmics: Scalable, dependency-aware task scheduling on heterogeneous manycores. *arXiv preprint arXiv:1606.04282*, 2016.

[46] Spyros Lyberis, Polyvios Pratikakis, Dimitrios S Nikolopoulos, Martin Schulz, Todd Gamblin, and Bronis R de Supinski. The myrmics memory allocator: hierarchical, message-passing allocation for global address spaces. *ACM SIGPLAN Notices*, 47(11):15–24, 2013.

[47] Milo M K Martin, Mark D Hill, and Daniel J Sorin. Why on-chip cache coherence is here to stay. *Commun. ACM ()*, 55(7):78–89, 2012.

[48] Alain Mebsout. *Inférence d'invariants pour le model checking de systèmes paramétrés*. PhD thesis, Université Paris Sud-Paris XI, 2014.

[49] Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli. Compiler testing via a theory of sound optimisations in the c11/c++ 11 memory model. In *ACM SIGPLAN Notices*, volume 48, pages 187–196. ACM, 2013.

[50] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Grappa: A latency-tolerant runtime for large-scale irregular applications. In *International Workshop on Rack-Scale Computing (WRSC w/EuroSys)*, 2014.

[51] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *USENIX Annual Technical Conference*, pages 291–305, 2015.

[52] Jarek Nieplocha and Bryan Carpenter. Armci: A portable remote memory copy library for distributed array libraries and compiler run-time systems. In *International Parallel Processing Symposium*, pages 533–546. Springer, 1999.

[53] Jaroslaw Nieplocha, Robert J Harrison, and Richard J Littlefield. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10(2):169–189, 1996.

[54] Brian Norris and Brian Demsky. Cdschecker: checking concurrent data structures written with c/c++ atomics. In *ACM SIGPLAN Notices*, volume 48, pages 131–150. ACM, 2013.

[55] Robert W Numrich and John Reid. Co-array fortran for parallel programming. In *ACM Sigplan Fortran Forum*, volume 17, pages 1–31. ACM, 1998.

[56] OpenMP Architecture Review Board. OpenMP application program interface version 4.5, 2015.

[57] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-tso. In *In TPHOLs'09: Conference on Theorem Proving in Higher Order Logics, volume 5674 of LNCS*, pages 391–407. Springer, 2009.

[58] Oded Padon, Kenneth L McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: safety verification by interactive generalization. *ACM SIGPLAN Notices*, 51(6):614–630, 2016.

[59] Ioannis Papadopoulos, Nathan Thomas, Adam Fidel, Nancy M Amato, and Lawrence Rauchwerger. Stapl-rts: An application driven runtime system. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 425–434. ACM, 2015.

[60] Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. Hierarchical task-based programming with StarSs. *International Journal on High Performance Computing Architecture*, 23(3):284–299, 2009.

[61] Antoniu Pop and Albert Cohen. Openstream: Expressiveness and data-flow compilation of openmp streaming programs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):53, 2013.

[62] Jonathan Protzenko. *Mezzo: a typed language for safe effectful concurrent programs*. PhD thesis, Université Paris Diderot-Paris 7, 2014.

[63] Scott Schneider, Christos D Antonopoulos, and Dimitrios S Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In *Proceedings of the 5th international symposium on Memory management*, pages 84–94. ACM, 2006.

[64] Sangmin Seo, Junghyun Kim, and Jaejin Lee. Sfmalloc: A lock-free and mostly synchronization-free dynamic memory allocator for manycores. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 253–263. IEEE, 2011.

[65] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, et al. Ott: Effective tool support for the working semanticist. *Journal of functional programming*, 20(1):71–122, 2010.

[66] Christi Symeonidou, Polyvios Pratikakis, Angelos Bilas, and Dimitrios S Nikolopoulos. Drasync: distributed region-based memory allocation and synchronization. In *Proceedings of the 20th European MPI Users' Group Meeting*, pages 49–54. ACM, 2013.

[67] M. B. Taylor. The raw processor specification. Technical report, 1999.

[68] William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In *International Conference on Compiler Construction*, pages 179–196. Springer, 2002.

[69] Sean Treichler, Michael Bauer, and Alex Aiken. Realm: An event-based low-level runtime for distributed memory architectures. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 263–276. ACM, 2014.

[70] Wayne Wolf, Ahmed Amine Jerraya, and Grant Martin. Multiprocessor system-on-chip (mpsoc) technology. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(10):1701–1713, 2008.

[71] Mani Zandifar, Nathan Thomas, Nancy M Amato, and Lawrence Rauchwerger. The stapl skeleton framework. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 176–190. Springer, 2014.