TRANSPARENT MANAGEMENT OF SCRATCHPAD MEMORIES IN SHARED MEMORY PROGRAMMING MODELS

Lluc Álvarez Martí

Barcelona, 2015

ADVISORS:

Xavier Martorell Bofill
Universitat Politècnica de Catalunya
Barcelona Supercomputing Center

Miquel Moretó Planas
Universitat Politècnica de Catalunya
Barcelona Supercomputing Center

Marc Casas Guix
Barcelona Supercomputing Center

A thesis submitted in fulfillment of
the requirements for the degree of
DOCTOR OF PHILOSOPHY / DOCTOR PER LA UPC

Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya

Abstract

Cache-coherent shared memory has traditionally been the favorite memory organization for chip multiprocessors thanks to its high degree of programmability. In this organization the cache hierarchy is in charge of moving the data and keeping it coherent between all the caches in the system, enabling the usage of shared memory programming models where the programmer does not need to carry out any data management operation. Unfortunately, performing all the data management operations in hardware causes severe problems, being the primary concerns the power consumption originated in the caches and the amount of coherence traffic in the interconnection network.

A good solution to these problems is to introduce ScratchPad Memories (SPMs) alongside the cache hierarchy, forming a hybrid memory hierarchy. SPMs are more power-efficient than caches and do not generate coherence traffic, but they degrade programmability. In particular, SPMs require the programmer to partition the data, to program data transfers, and to keep coherence between different copies of the data.

A promising solution to exploit the benefits of the SPMs without introducing programmability difficulties is to allow the programmer to use shared memory programming models and to automatically generate code that manages the SPMs. Unfortunately, current compilers and runtime systems encounter serious limitations to automatically generate code for hybrid memory hierarchies from shared memory programming models.

This thesis proposes to transparently manage the SPMs of hybrid memory hierarchies in shared memory programming models. In order to achieve this goal this thesis proposes a combination of hardware and compiler techniques to manage the SPMs in fork-join programming models and a set of runtime system techniques to manage the SPMs in task programming models. The proposed techniques allow to program hybrid memory hierarchies with these two well-known and easy-to-use forms of shared memory programming models, capitalizing on the benefits of hybrid memory hierarchies in terms of power consumption and network traffic without harming the programmability of the architecture.

The first contribution of this thesis is a hardware/software co-designed coherence protocol to transparently manage the SPMs of hybrid memory hierarchies in fork-join programming models. The solution allows the compiler to always generate code to manage the SPMs with tiling software caches, even in the presence of unknown memory aliasing hazards between memory references to the SPMs and to the cache hierarchy. On the software side, the compiler generates a special form of memory instruction for memory references with possible aliasing hazards. On the hardware side, the special memory instructions are diverted to the correct copy of the data using a set of directories that track what data is mapped to the SPMs.

The second contribution of this thesis is a set of runtime system techniques to manage the SPMs of hybrid memory hierarchies in task programming models. The proposed runtime system techniques exploit the characteristics of these programming models to map the data specified in the task dependences to the SPMs. Different policies are proposed to mitigate the communication costs of the data transfers, overlapping them with other execution phases such as the task scheduling phase or the execution of the previous task. The runtime system can also reduce the number of data transfers by using a task scheduler that exploits data locality in the SPMs. In addition, the proposed techniques are combined with mechanisms that reduce the impact of fine-grained tasks, such as hardware runtime systems or large SPM sizes.

The forthright accomplishment of the contributions of this thesis is that hybrid memory hierarchies can be programmed with fork-join and task programming models. Consequently, architectures with hybrid memory hierarchies can be exposed to the programmer as a shared memory multiprocessor, taking advantage of the benefits of the SPMs in power consumption and network traffic while maintaining the programming simplicity of shared memory programming models.

Contents

Chapter 1

Introduction

The evolution of microprocessors design has changed significantly in the last few decades. For many years the Moore's law drove the progress of every new generation of processors by increasing the number of transistors in the chip and its clock frequency, allowing to build faster and more complex single-core processors that could exploit the Instruction Level Parallelism (ILP) of sequential programs. As shown in Figure 1.1, this trend continued until the early 2000s, when the Moore's law encountered two fundamental obstacles: the ever increasing latency of memory accesses due to the speed gap between the processor and the main memory, known as the Memory Wall [156], and the ever increasing power consumption of chips with higher number of transistors and clock rates, known as the Power Wall [105] or the end of Dennard's scaling [44].

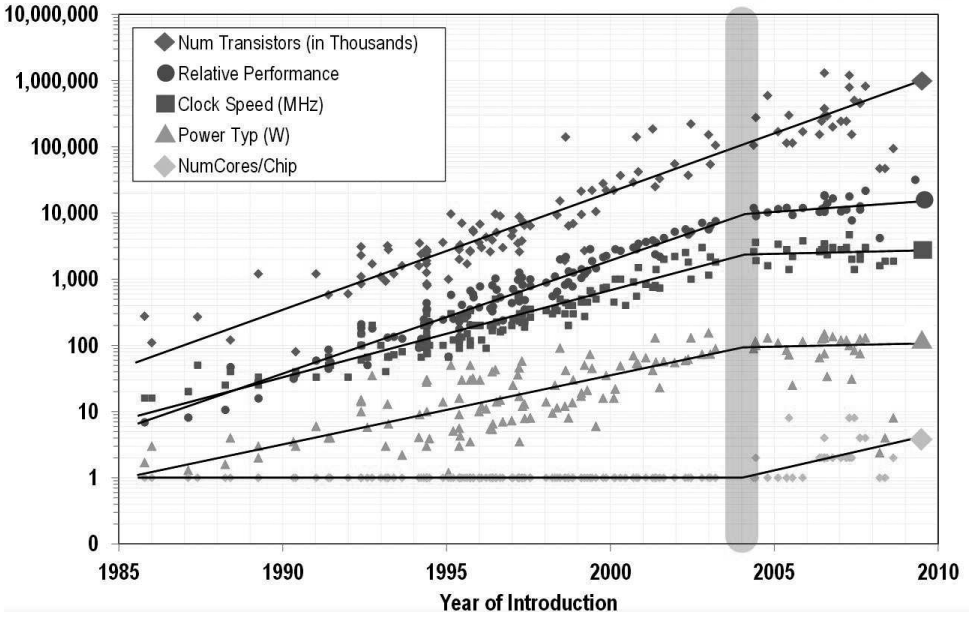

Figure 1.1: Evolution of microprocessors

In the early 2000s, to overcome the stagnation of single-core processors' performance, the landscape of microprocessors design entered the multicore era. Multicore processors can potentially provide the desired performance gains by exploiting the Task Level Parallelism (TLP) of parallel programs, but they have to face problems concerning the orchestration of parallel workloads such as the communication between cores or the concurrency in memory accesses. For this reason, a key element of multicore designs is the memory hierarchy that connects the different computing elements in the chip.

Cache-coherent shared memory has traditionally been the favorite on-chip memory hierarchy for multicore processors. The major reason for the success of this approach is its high programmability, which is achieved by moving data and keeping it coherent between all the caches of the system at the microarchitecture level, without any intervention from the programmer. This is accomplished by a hierarchy of caches, that exploit locality to keep the data close to the core that is accessing it, and by a cache coherence protocol, which manages the validity of the contents of all the caches in the chip to ensure that memory accesses are always served with non-stale values. Unfortunately, the cost of performing these actions in hardware becomes an obstacle to scale up the number of cores [136], being the primary concerns the power consumption originated in the caches and the amount of coherence traffic in the interconnection Network on-Chip (NoC) that is required to maintain all the data in a coherent state.

ScratchPad Memories [14] (SPMs) are a well-known alternative to cache hierarchies in domains where power-efficiency is of paramount importance. SPMs are simple pieces of memory with their own address space, directly addressable by the software and incoherent with the rest of memories in the chip. These memories are less complex than caches because they do not implement any mechanism to automatically request data or to track the state of the data that is being kept, so they can serve memory accesses as fast as caches but in a much more power-efficient way and without originating any coherence traffic. However, SPMs suffer from poor programmability as they require the software to explicitly transfer data between the SPMs and any other memory in the system and to keep coherence between different copies of the data in different address spaces.

The trend in the High Performance Computing (HPC) domain towards massively parallel manycore processors makes not possible to keep relying neither in purely cache-coherent memory hierarchies, due to their power consumption and scalability issues, nor in pure SPMs designs, due to their programmability issues. Instead, computer architecture for HPC is exhibiting a trend towards more heterogeneity, which has been shown

effective in architectures such as the Cell B. E. [79] or GPGPUs [66]. These designs have different kinds of cores and hybrid memory hierarchies that combine caches and SPMs to provide performance under an affordable power budget, but the programmability issue is still not solved [136]. Processors with hybrid memory hierarchies have very complex memory models that move away from the shared memory paradigm, which imposes serious limitations for the widespread usage of these systems. First, redefining the memory model breaks backwards compatibility, so every new architecture requires adapting scientific and industrial codes. Second, exposing deep and hybrid memory hierarchies to the programmer significantly degrades programmability, as the programmer needs to partition the data, explicitly transfer data between memory spaces, and handle potential data replications. Giving these responsibilities to the programmer complicates the process of writing the code and, more importantly, requires the programmer to have advanced knowledge of the architecture to perform the data management operations efficiently.

A promising solution to solve the programmability issues of hybrid memory hierarchies is to allow the programmer to use well-known shared memory programming models and to automatically generate code that manages the memory hierarchy. Shared memory programming models have been widely used since the first multicore architectures appeared, and they are easy to use because they rely on simple memory models exposed by multicores with cache-coherent memory hierarchies, so the programmer does not need to explicitly manage the data. Unfortunately, current compilers and runtime systems encounter serious limitations to automatically generate code for hybrid memory hierarchies from shared memory programming models.

Fork-join models like OpenMP [114] are the most traditional form of shared memory programming models. In these models sequential execution phases are interleaved with parallel phases that are specified by the programmer at designated points of the program. It has been shown in the past that a good way to manage SPMs in fork-join models is to use tiling software caches [55, 56, 67, 133]. Tiling software caches take advantage of the predictability of strided memory accesses to efficiently map them to the SPMs, while unpredictable random memory accesses are served by the cache hierarchy. Even though compilers succeed in transforming code to use tiling software caches when the computation is based on predictable memory access patterns, in the presence of unpredictable memory accesses they encounter important limitations [55, 56, 67]. Due to the incoherence between the SPMs and the cache hierarchy, the compiler cannot generate code for the SPMs if it cannot ensure that there is no aliasing between two memory references that

may target copies of the same data in the SPMs and in the cache hierarchy. This memory aliasing problem greatly restricts the ability of the compiler to generate code for hybrid memory hierarchies in non-trivial cases.

The expected heterogeneity of future manycore architectures has caused that, in recent years, task programming models such as the task extensions introduced in OpenMP 4.0 [115] have emerged. In these programming models the programmer exposes the available parallelism of an application by splitting the code in sequential pieces of work, called tasks, and by specifying the data and control dependences between them. With this information the runtime system manages the parallel execution of the workload following a data-flow scheme, scheduling tasks to cores and taking care of synchronization between tasks. In addition to the programmability advantages for complex heterogeneous architectures, a very important benefit of task programming models is that decoupling the application from the architecture also allows to take advantage of the available information in the runtime system to drive optimizations in a generic and application-agnostic way [146, 28, 119, 100, 101, 61]. Following this trend, a promising solution to manage the SPMs of hybrid memory hierarchies without affecting the programmability of the architecture is to adopt task programming models and to give the runtime system the responsibility of exploiting task annotations to map the data specified in the task dependences to the SPMs, so memory accesses to this data are served more efficiently during the execution of the tasks.

1.1 Thesis Objectives and Contributions

The main goal of this thesis is to propose a combination of hardware, compiler and runtime system techniques to manage the SPMs of hybrid memory hierarchies from shared memory parallel programming models, transparently to the programmer.

The contributions of this dissertation settle the most important limiting factors for the widespread usage of hybrid memory hierarchies. The forthright benefit of the proposed techniques is that they allow architectures with hybrid memory hierarchies to be programmed with well-known and easy-to-use shared memory parallel programming models, to keep the simplicity of memory models exposed by cache-coherent shared memory multiprocessors, and to preserve code portability. This allows to capitalize on the benefits of hybrid memory hierarchies in terms of power consumption and NoC traffic without affecting the programmability of the architecture.

In order to achieve these goals, this thesis proposes two different approaches for the automatic management of SPMs in the two most prominent forms of shared memory programming models: fork-join and task models.

1.1.1 Automatic Management of SPMs in Fork-Join Models

The first contribution of this thesis is a hardware/software co-designed coherence protocol that allows the compiler to always generate code to manage the SPMs of hybrid memory hierarchies in fork-join programming models, even if it encounters memory aliasing hazards between memory references to the SPMs and memory references to the cache hierarchy. On the software side, the proposed solution consists of simple modifications to the compiler analyses so that it can classify memory references in three categories: strided memory references, random memory references that do not alias with strided ones, and random memory references with unknown aliases. The compiler then transforms the code for the strided memory references to map them to the SPMs using tiling software caches while, for the random memory references that do not alias with strided ones, it generates memory instructions that access the address space of the cache hierarchy. For the random memory references with unknown aliasing hazards the compiler generates a special form of memory instruction that gives the hardware the responsibility to decide what memory is used to serve them. On the hardware side, a coherence protocol is proposed so that the architecture can serve the memory accesses with unknown aliasing hazards with the memory that keeps the valid copy of the data. For this purpose the hybrid memory hierarchy is extended with a set of directories and filters that track what part of the data set is mapped and not mapped to the SPMs. These new elements are checked at the execution of memory accesses with unknown aliases to divert them to the correct copy of the data, so all memory accesses can be correctly and efficiently served by the appropriate memory.

1.1.2 Automatic Management of SPMs in Task Models

The second contribution of this thesis is a set of techniques for runtime systems of task programming models to transparently manage the SPMs of hybrid memory hierarchies. The proposed extensions in the runtime system exploit the characteristics of task programming models to manage the SPMs very efficiently. Since the data dependences of a task are specified in the source code, the runtime system is aware of what data is going to be accessed by the task before it is executed, so it can exploit this information to map

the task dependences to the SPM of the core that is going to execute the task. In addition, the runtime system can take advantage of the execution model of task programming models to hide the communication costs of the data transfers required by the SPMs, either by overlapping them with other execution phases or by exploiting data locality in the task scheduling decisions. The proposed techniques also take into account that the size of the SPMs has a direct impact on the granularity of the tasks, so the study contemplates pairing the proposed SPM management strategies with different SPM sizes and task runtime systems with hardware support, analyzing the trade-offs of each solution in terms of performance, power consumption and NoC traffic.

1.2 Thesis Structure

The contents of this thesis are organized as follows:

Chapter 2 reviews the state of the art in cache-coherent shared memory multiprocessors and in shared memory programming programming models, exposing their main characteristics and problems. Then it describes the existing multiprocessors with hybrid memory hierarchies, explaining their architectural details, how each of them is programmed, and highlighting their programmability issues.

Chapter 3 introduces the simulation infrastructures and software environments used to perform the experiments described in this thesis, as well as the benchmarks and the metrics used for the evaluation of the proposals.

Chapter 4 presents the hardware/software co-designed coherence protocol to transparently manage the SPMs of hybrid memory hierarchies in fork-join programming models. The chapter first explains the internals of tiling software caches and the coherence problem they expose to then describe the compiler support and the hardware extensions of the proposed solution.

Chapter 5 proposes to manage the SPMs of hybrid memory hierarchies from the runtime system of task programming models. The chapter highlights the suitability of task programming models for SPMs, describes the extensions in the runtime system to manage them and analyses several techniques to minimize the overheads of the runtime system and the data transfers.

Chapter 6 concludes this dissertation by remarking its main contributions and by providing a brief summary of the future work.

Chapter 2

State of the Art

This chapter gives an overview of the state of the art in the area of chip multiprocessors and parallel programming models. The first section reviews shared memory multiprocessors, explaining the most relevant shared memory programming models that are used to program them, the cache-coherent shared memory organization that is used to support these programming models, the inefficiencies of current cache hierarchies, and the solutions proposed in the literature. The second section analyzes multiprocessors that incorporate SPMs in their memory hierarchies, discussing the architectural details of the different memory organizations, their implications on the programmability of the architecture, and the techniques proposed to automatically manage the SPMs without the intervention of the programmer.

2.1 Shared Memory Multiprocessors

Shared memory multiprocessors are the most genuine representatives of chip multiprocessors. This family embraces a wide number of chips, from the first commercial multicore processors that emerged in the early 2000s such as the IBM POWER4, the Intel Core Duo or the AMD Opteron, to current high-end manycore architectures for HPC such as the Intel Xeon Phi or the IBM Blue Gene/Q. The distinctive characteristic of shared memory multiprocessors is its memory organization, composed by a hierarchy of caches with a cache coherence protocol. This scheme allows the different cores to share data without any intervention from the programmer, enabling the usage of shared memory programming models to program the architecture.

2.1.1 Shared Memory Programming Models

Shared memory programming models offer a global virtual address space to allow multiple tasks to implicitly share data. These programming models have been the preferred option to program shared memory multiprocessors since they first appeared. Compared to other parallel programming paradigms, the shared memory approach is usually considered more productive, more intuitive, easier to learn, and easier to use, since it releases the programmer from the burden of explicitly distributing the data set of the application among tasks and explicitly communicating data between them. Instead, data is shared in the global address space and can be accessed by any task at any time and from any core. In addition, shared memory programming models are supported in most multicore and manycore architectures, so the same code can be executed in a broad range of processors. Although some tuning in the code is required to get the maximum performance of each particular architecture, code portability is a major advantage for the community as it drastically reduces development costs.

2.1.1.1 Thread Libraries

The most basic representatives of shared memory programming models are the thread libraries offered by the Operating Systems (OSs). These libraries are used to expose the capabilities of shared memory multiprocessors to the software, offering low level functionalities to create threads, assign tasks to threads, and synchronize them. Examples of thread libraries are POSIX threads [29] as implemented in Linux [106], a slight variation for SUN Solaris platforms called Solaris threads [140], or the Win32 threads library [23] available in the different versions of Microsoft Windows.

Some programming languages offer thread libraries with similar characteristics, aiming to provide a low level thread interface that is portable across OSs. For instance, Java threads [111] are included as part of the standard Java Development Kit, and C++ [139] includes classes for thread management since its revision from 2011.

2.1.1.2 OpenMP

OpenMP is the most commonly used programming model for shared memory multiprocessors. OpenMP allows to specify parallel constructs in C, C++ and Fortran using simple and portable compiler directives. These directives are supported by the vast majority of modern compilers and OSs for shared memory multiprocessors.

The core elements of OpenMP are the directives to specify parallel regions, workload distribution, data-environment management and thread synchronization. OpenMP uses a fork-join parallel execution model, where a single thread is used in sequential regions and the execution branches off in parallel at designated points in the program, specified by the programmer with the `#pragma omp parallel` directive. OpenMP also allows to specify how the work is distributed between threads in a parallel region, either assigning independent blocks of code to each thread using the `#pragma omp section` directive or distributing the loop iterations among threads using the `#pragma omp for` directive. The latter form is the one that exploits loop parallelism, which is the most commonly used parallelization pattern in OpenMP programs. In this model the `schedule` directive can be used to control how the iterations are distributed between the threads: a `static` assignment of iterations to threads, a `dynamic` scheme where threads request fixed-size blocks of loop iterations from a work queue, or a `guided` approach where threads dynamically request variable-size blocks of iterations. In addition, OpenMP allows to specify the sharing attributes of the data in any parallel region, that can be `shared` or `private` between threads, and supports synchronization directives such as `critical` for critical sections, `atomic` to update values in memory atomically or `barrier` to synchronize all threads in a point of the program.

The directives introduced by the programmer are processed by the compiler to generate parallel code. In this process the compiler arranges the code so that the parallel regions are encapsulated in separate functions, it sets up the declaration of the variables according to its sharing attributes, and it adds function calls to the runtime system in the points of the code where the parallelism is forked, joined and synchronized. When the code is executed the runtime system is in charge of managing the threads and the shared variables. For this purpose the runtime system provides routines to create threads, synchronize them, and destroy them. In order to assign tasks to threads the runtime system implements schedulers and work queues that support all the forms of parallelism allowed by the OpenMP directives. In addition, the runtime system maintains the versions of the shared variables as well as other internal control variables.

This coordinated effort between the compiler and the runtime system is what allows OpenMP to generate parallel code and to manage the parallel execution from simple directives. This model has been very successful because it allows to exploit the capabilities of shared memory multiprocessors with a programming interface that is easy to use for programmers and portable across many architectures and systems.

2.1.1.3 Task Programming Models

Several task programming models have emerged in recent years to face the expected complexity of future heterogeneous multicore and manycore architectures. These programming models conceive the execution of a parallel program as a set of tasks with dependences among them.

In task programming models the programmer only has to split the serial code in tasks and to specify the dependences between the tasks. With this information the runtime system manages the parallel execution of the tasks, taking care of scheduling tasks to cores and synchronizing them without any intervention from the programmer. In order to manage the execution of the tasks the runtime system constructs a Task Dependence Graph (TDG), which is a directed acyclic graph where the nodes are tasks and the edges are dependences between them. Similarly to how an out-of-order processor schedules instructions, the runtime system schedules a task on a core when all its input dependences are ready and, when the execution of the task finishes, its output dependences become ready for the next tasks.

Task data-flow programming models are a subcategory of task programming models where the task dependences are represented with the data sets accessed by each task. In these models the programmer specifies the tasks and its associated data sets, which define three types of dependences: input dependences are memory regions read by a task, output dependences are memory regions written by a task, and input-output dependences are memory regions both read and written by a task. Many task data-flow programming models have been proposed recently. OpenMP 3.0 [9] provides compiler directives to support basic tasking constructs, that are extended with data dependences in OpenMP 4.0 [115], while OmpSs [51] extends OpenMP 4.0 with additional directives to specify task priorities and special tasking constructs. Other task programming models add language extensions to specify tasks and its associated data. The Codelets model [164] breaks applications into tasks with data and control dependences, and relies on this decomposition to improve load balancing and reduce data motion. Unlike in OpenMP 4.0, in Codelets the programmer needs to explicitly specify not only data dependences but also the particular codelet each dependence is associated with. StarPU [8] is a C API that allows to declare tasks and assign functions and data to them. Legion [16] programs are decomposed in tasks that access data partitions manually specified by the programmer, while in Sequoia [58] tasks have their private address space and it is the programmer who organizes them hierarchically. Charm++ [80] is a C++ based asynchronous message driven programming

model where the programmer decomposes a program into message-driven objects called chares. Chares are distributed among the processors and the runtime system is in charge of sending messages to the chares to trigger the execution of the code within them. The Habanero [134] project proposes language extensions to C and Java to create tasks, express data locality and explicit communication between tasks.

Other task programming models allow to decompose a program in tasks, but they are not data-flow since tasks do not have data dependences between them. Intel TBB [126] is a C++ template library that implements a task execution model where the programmer splits the serial code into tasks that have implicit control dependences with their parents and children tasks. Cilk [26] extends C and C++ with keywords to spawn and synchronize tasks, and uses a simple fork-join model enhanced with work-stealing primitives to balance the load efficiently. Although it is not part of the standard language, Vandierendonck et al. [150] propose to extend Cilk with data dependences between tasks to ease programming parallel patterns such as pipelines. These extensions would turn Cilk into a task data-flow programming model.

The characteristics of task programming models make them very suitable for current and future heterogeneous architectures. The main reason is that these models allow the programmer to specify parallelism in an architecture-agnostic way, hiding to the programmer the complex architectural details and enabling the same code to be executed in very different platforms. In addition, the data dependences of task data-flow programming models can be used to program not only shared memory multiprocessors but also heterogeneous architectures that require explicit data transfer between address spaces, such as architectures with SPMs. Section 2.2 explains the most representative architectures with SPMs and also describes how task data-flow programming models are used to manage the memory hierarchy of some of these architectures.

2.1.2 Cache-Coherent Shared Memory Hierarchies

The cache-coherent memory hierarchy found in shared memory multiprocessors is the key element to implement a single address space where all the cores in the architecture can efficiently share data, which is the fundamental requirement to support shared memory programming models. This memory organization is composed of a set of cache memories organized hierarchically and a cache coherence protocol.

2.1.2.1 Cache Memories

A fundamental part of cache-coherent shared memory hierarchies are the cache memories, or simply caches. Caches are pieces of memory that keep data close to the CPU that is accessing it. Figure 2.1 shows a scheme of a modern cache organization. A cache is a collection of cache blocks that contain the data, flag bits, and the tag of the block, which contains part of the address to which the data belongs to. Caches organize the blocks in different ways, that are accessed in parallel when a cache block is requested. In addition, since caches are self-managed, they also implement replacement algorithms to automatically allocate and evict cache blocks.

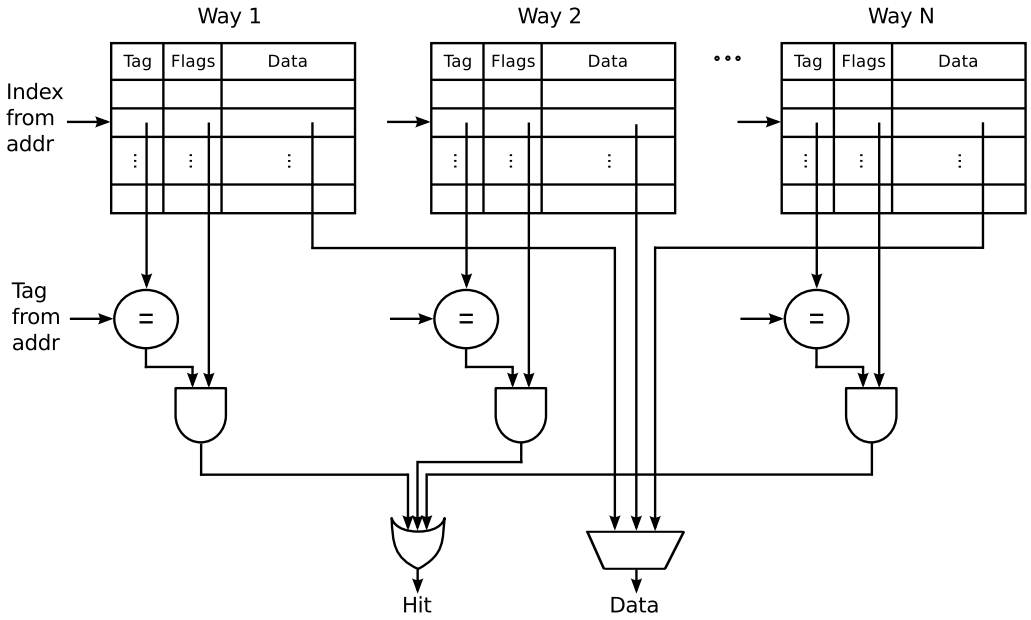

Figure 2.1: Cache memory organization

Shared memory multiprocessors incorporate multiple caches in a hierarchical organization. Current cache hierarchies usually implement two or thee levels of caches, where the first level uses small and fast caches and subsequent lower levels use bigger and slower caches. For example, the processors based on the Intel Broadwell microarchitecture have a cache hierarchy of three levels, where each core has separate L1 caches for data and instructions, each of 32 KB and 8-way associative, and an 8-way associative L2 cache of 256 KB, and all cores share a L3 cache of up to 6 MB that is 16-way associative. On the other hand, the Intel Xeon Phi uses a cache hierarchy of two levels where every core has separate 8-way associative L1 caches of 32 KB for data and instructions and an 8-way associative L2 cache of 512 KB.

2.1.2.2 Cache Coherence Protocol

The cache coherence protocol is the mechanism that ensures that changes in the values of shared operands are propagated throughout the cache hierarchy. As a result, every core observes the last produced value and never consumes stale data. Current shared memory multiprocessors generally implement directory-based cache coherence protocols. In this scheme the data held in the caches is tracked by a cache directory that maintains coherence between all the caches.

The cache directory of the cache coherence protocol records the state of each cache block. As shown in Figure 2.2, the cache directory keeps, for each cache block, its state and a bit vector of sharer cores that tracks which cores have the cache block in its cache. Current shared memory multiprocessors typically use the MOESI cache coherence protocol, which uses five states for the cache blocks: modified (M), owned (O), exclusive (E), shared (S) and invalid (I). The cache directory acts as a manager to which the cores ask permission to request or write back a cache block and notify evictions and updates of a cache block. When the cache directory receives a message from a core it performs the actions defined in the cache coherence protocol to ensure coherence. Depending on the type of the request and the state of the cache block, the actions may involve reading data from memory or from some other cache, changing the state of the cache block, updating its sharer cores, invalidating cache blocks in other caches, etc.

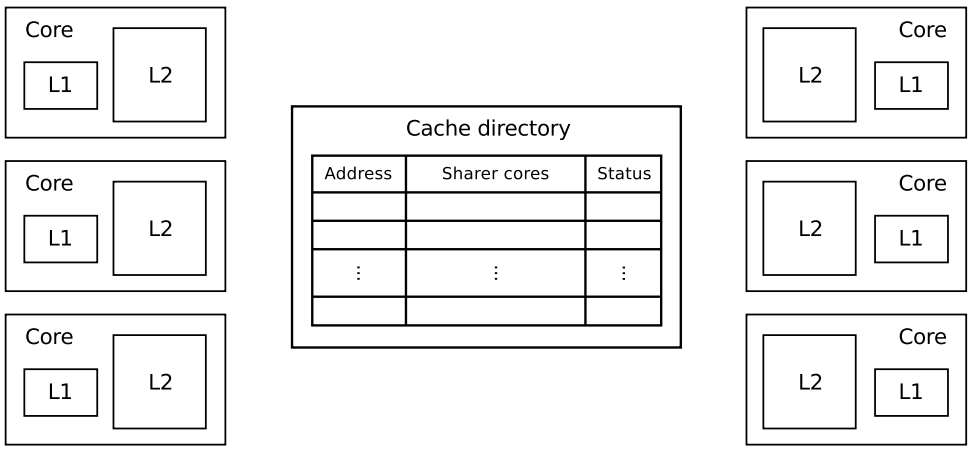

Figure 2.2: Directory-based cache coherence protocol

2.1.3 Inefficiencies of Cache Hierarchies

Cache-coherent memory hierarchies present many inefficiencies, specially when a large number of cores is integrated in the architecture. The primary concerns are the power consumption originated in the caches and the amount of NoC traffic generated by the cache coherence protocol to keep all the data in a coherent state.

2.1.3.1 Power Consumption

The power consumption originated in the cache hierarchy is one of the most important problems of current shared memory multicore architectures. Caches dissipate a significant fraction of the overall power budget of the whole chip, up to 45% in current manycore architectures [136], due to their hardware complexity and their high utilization. In order to alleviate this problem a vast number of techniques have been proposed to reduce both the static and the dynamic power consumed in the caches.

Numerous works aim to save static power (also called leakage power) by powering down or off parts of the cache. Powell et al. [122] propose a Gated-Vdd technique to power off unused cache blocks, which drastically reduces their leakage power at the cost of losing their contents. Alternatively, Flautner et al. [59, 60] propose Drowsy Caches, which power down unused cache blocks by putting them in drowsy mode (a kind of sleep mode) to reduce their leakage power without losing their contents. Agarwal et al. [2] propose a gated-ground scheme that allows powering off cache blocks but still preserving their contents at the cost of minor performance and area overheads. The problem of these mechanisms is that, when a cache block is accessed, a cache miss occurs if the block is powered down or off, even if the cache has the block in valid state. Many heuristics have been proposed to overcome this problem, deciding when and which parts of the cache can be deactivated with minimal performance degradations.

Some heuristics aim to deactivate cache blocks individually. Cache Decay [83, 84] cuts off the power supply of cache blocks if a pre-set number of cycles have elapsed since their last access. Zhou et al. [162, 163] propose to deactivate only the data array of cache blocks so the tag array can be accessed to calculate the increase of the cache miss rate caused by turning off blocks, allowing fine-grained control of the performance losses and to re-activate cache blocks if needed. Drowsy Instruction Caches [87] monitor unconditional branches and subroutine calls and returns to predict and selectively wake up only the blocks that are going to be accessed in the instruction cache, allowing most

of the cache to stay in drowsy mode without significant performance losses. Li et al. [96] propose to deactivate L2 cache blocks when their data also exists in the L1 cache.

Other works propose more aggressive heuristics to deactivate complete cache ways. Albonesi [3] proposes Selective Cache Ways, a cache design that enables only a subset of the ways when the cache requirements of the running applications are low. In this design it is the software who explicitly turns on and off the cache ways using special instructions. C. Zhang et al. [157, 158] show that a very similar technique can be utilized in embedded processors, and W. Zhang et al. [160] propose two compiler algorithms to apply the same idea to instruction caches. Totoni et al. [144] use formal language theory to identify program phases and propose a runtime system that turns off cache ways at the start of each phase. Without any software support, Balasubramonian et al. [13] propose to detect program phases in the microarchitecture and to monitor the CPI, cache and TLB usage of each phase to turn off whole cache ways according to its requirements. Dhodapkar and Smith [47, 48] refine this approach by introducing a hardware structure that saves the appropriate configuration for each program phase.

The dynamic energy dissipated in the caches is another important concern in modern processors. This problem has also been widely studied, and various techniques have been proposed to alleviate it. In general, these approaches try to minimize the number of operations or the amount of data read and written on every cache access.

Way prediction is a technique that has been proposed to reduce the dynamic power consumption of caches. It consists on predicting which way of the cache will contain the data and accessing only the predicted way instead of all the ways in parallel. Way prediction was initially proposed by Calder et al. [30] to reduce the access time of sequential associative caches, analyzing different sources for the prediction such as the effective address, the registers used in the address calculation or the program counter of the instruction. Batson and Vijaykumar [15] show that very similar way prediction techniques can also achieve significant savings in dynamic energy. Inoue et al. [75] propose to keep the most recently accessed way to perform the prediction, and Powell et al. [123] show that combining way prediction in L1 instruction caches and selective direct-mapping in L1 data caches can further reduce the dynamic energy of the first level of the cache hierarchy. The main problem of way prediction is that it adds performance penalties when the accessed way is misspredicted and, moreover, it is mainly proposed for L1 caches because they present very predictable access patterns but, for lower level caches, it is not very effective since locality is hidden by previous cache levels.

Instead of prediction, way caching records the way of recently accessed cache blocks to reduce the dynamic energy of highly associative caches. Min et al. [102] propose to use a small Way Cache, a cache that stores the way accessed by previously observed cache accesses. If there is a hit in the Way Cache the data cache is accessed as a direct-mapped cache, otherwise it is referenced as a conventional associative cache. Nicolaescu et al. [109] propose to add coherence between the Way Cache and the data cache, invalidating Way Cache entries when blocks are evicted or invalidated from the data cache. This feature makes the contents of the Way Cache always correct, so the outcome is not a prediction but a determination. An important problem of these two works is that the Way Cache is accessed before the cache access, adding delays to the critical path of data accesses. Zheng et al. [161] propose Tag Check Elision, which determines the correct cache way early in the pipeline by doing a simple bounds check to decide if a memory access is to the same block as an earlier access. The bounds check occurs as soon as the virtual address is available, so it does not add any pipeline delay or performance degradation, and the TLB access can also be eliminated in a physically-tagged cache.

Other techniques are proposed to filter unnecessary accesses to cache ways. Zhang et al. [159] propose the Way-Halting Cache, that compares the four least significant bits of the tag during the index decoding to filter way accesses. This approach performs a fully-associative search in the first comparison which negatively affects power consumption. Ghosh et al. [65] propose Way Guard, a mechanism for large associative caches that employs bloom filters to reduce dynamic energy by skipping the lookup of cache ways that do not contain the requested data according to the bloom filter. This scheme requires the addition of a large decoder and two fields per way: one segmented bloom filter, previously proposed by the same authors to filter accesses to the whole cache [64], and another bloom filter to filter accesses to ways. Although Way Guard shows performance gains with respect to the Way-Halting Cache, it suffers from higher overheads in terms of area and adds significant complexity. The main problem of these techniques is that a new structure must be accessed serially after the address translation in the TLB and before accessing the cache, hence either increasing cycle time or adding an extra cycle to the memory access latency in virtually-indexed physically-tagged caches. In order to overcome this problem, Valls et al. [147, 148] propose PS-Cache, a mechanism that filters the ways looked up on each cache access by classifying each block as private or shared, according to the page table information. On a cache access, only the ways containing blocks that match with the classification of the requested block are searched. The same authors also propose the

Tag Filter Cache [149], a cache architecture that uses the least significant bits of the tag part of the address to discern which ways may contain the accessed block.

Including way information in the TLB is another approach to reduce the energy consumed in the caches. The goal is to prevent doing one lookup in the tag array of the cache to determine the way of the cache block and another lookup in the TLB to provide the address translation. Instead, the idea is to do a single lookup in the TLB that provides both the address translation and the way where the block is located in the cache. In order to do so each TLB entry is extended with a field that contains, for each cache block belonging to the page, the way where it is located in the cache. Boettcher et al. [27] propose to use this scheme to predict or determine the way of the accessed block in the L1 cache, but the way information in the TLB is not always deterministic, so it is sometimes treated as a hint and the tag array of the cache has to be kept. The Tag-Less Cache [132] introduces a valid bit along with the way information to make it deterministic, so the tag array of the cache can be eliminated. Although these two approaches achieve substantial energy gains, they require significant area to store the way information for each cache block of the page (specially if big page sizes are supported), they require changes in the cache replacement policy, they affect miss ratios and performance in unpredictable ways, and they need very costly reverse address translation mechanisms to handle cache coherence and virtual address synonyms because they use virtual indexing. An extension of the Tag-Less Cache [130, 131] uses a physically-tagged hub per core to handle virtual address synonyms and external coherence requests and, in addition, they store not only the way but also the cache where the block is kept, so the whole cache hierarchy can be navigated with a single lookup.

2.1.3.2 Coherence Traffic

Some research works propose to eliminate the unnecessary coherence traffic that is generated by memory accesses to private data. The technique consists on distinguishing what data is private and what data is shared, and deactivating the cache coherence protocol for the private data. The deactivation of the coherence protocol implies that cache hits to private data do not send invalidation coherence messages to the other caches, and cache misses to private data do not request the cache block to the other caches but are directly served by the main memory. The proposed techniques differ significantly one from each other, specially in the way the private data is identified, the granularity at which it is identified and the hardware mechanisms that keep track of the privateness.

Cuesta et al. [40, 41] propose to track private virtual memory pages at the OS level, extending the functionality of the page fault handler. When a page is accessed for the first time in an application, the OS sets a private bit in the page table and in the TLB of the core that is accessing it. Memory accesses from the same core to the private page are served without requiring any coherence messages to the caches nor to the cache directory of the cache coherence protocol. When another core accesses the page the OS marks the page as shared and triggers a flush of the cache blocks and the TLB entries of the page in the first core. Subsequent accesses by any core to the shared page trigger the coherence actions defined by the cache coherence protocol. In another paper [39], the authors extend this mechanism to work also for shared read-only pages. Almost identical data categorization methods and hardware mechanisms have been proposed to filter snoop requests [86] and to optimize the data placement in NUCA caches [71, 72]. Although this technique only requires small changes in the TLB and the OS, changing the state of the data from private to shared is very costly because it requires to flush the TLBs and the caches. Moreover, it does not eliminate coherence traffic completely because it suffers from false positives due to the granularity used to track the memory regions and because, even if some data is only shared in a small phase of the program, it never transitions from shared back to private.

Other works propose to track private data at the granularity of memory regions of arbitrary size that are managed at the microarchitecture level, transparently to any software layer. In these works new hardware structures are used to classify a memory region as private when a core first accesses it, and subsequent accesses by other cores to the same memory region make it shared. Cantin et al. [31, 32] add a separate hardware structure called the Region Coherence Array that directly interacts with an ad-hoc coherence protocol. RegionScount [103, 104] also identifies private and shared memory regions of arbitrary size at the microarchitecture level, and additional hardware support is used to filter broadcasts in snoop-based cache coherence protocols. Alisafaee [4] proposes to extend the cache directory of the cache coherence protocol with special entries to track private memory regions and, additionally, timing is taken into account to do the data classification. The reasoning is that, even if the data is shared, coherence is not needed if two cores do not access it at the same time. The first memory access by a core to a region makes it private, and subsequent accesses from the same core keep this categorization. A region is marked as shared when another core accesses it, and it is marked back as private when all the cache blocks of the region are evicted from the caches of all the cores but

one. On the one hand, the advantage of these mechanisms compared to the ones that use a fixed page size granularity with OS and TLB support is that they work transparently to the software, and the size of the memory regions can be dynamically adapted to improve the accuracy of the data classification. Moreover, transitioning from shared to private further improves accuracy, although relying on cache block evictions causes that the transition does not happen immediately when the data stops being shared. On the other hand, these techniques introduce significant hardware complexity and more power is consumed in the execution of memory instructions, as they need to access an additional hardware structure.

2.1.4 Future Directions

Although the combination of shared memory programming models and cache-coherent memory hierarchies has dominated the landscape of parallel computing for many years, the aforementioned power consumption and scalability problems are becoming a severe obstacle to scale up the number of cores in future shared memory manycore architectures [7, 136].

The different solutions discussed in the previous sections can reduce the power consumption or the NoC traffic in some cases, but they all have some limitations or introduce new difficulties. In addition, the solutions target very specific parts of the problem, so many of them have to be combined to make cache hierarchies power-efficient and scalable enough to fulfill the requirements of future shared memory manycore architectures. Combining the different techniques requires significant complexity, and even some solutions are incompatible between themselves, so it is unclear that these approaches will end up solving the problems of cache hierarchies in the long term.

A more disruptive way to solve the inherent inefficiencies of cache hierarchies is to move away from the pure shared memory paradigm [1, 7]. The main reasoning behind this approach is that, fundamentally, doing an efficient management of the data of a parallel application is a very challenging problem, and solving it at the microarchitecture level is very costly. Instead, giving the software layers the responsibility of performing certain data management operations opens the door to designing much more efficient memory hierarchies.

2.2 Multiprocessors with Scratchpad Memories

Many commercial processors and research projects have adopted memory organizations different than cache-coherent shared memory hierarchies. These different memory organizations are frequent in the embedded domain, where processors have strong power consumption constraints, and in the HPC domain, where power efficiency and scalability are of paramount importance.

The key element of the efficient memory hierarchies used in the embedded and the HPC domain are the ScratchPad Memories (SPMs). SPMs are a simple arrays of memory exposed to the software layers in a part of the virtual and physical address spaces. Compared to caches, SPMs provide three main advantages. First, SPMs do not need a tag array, a hit/miss logic, nor a replacement policy logic, so they are more power-efficient and they occupy less area than caches. Second, memory accesses to the SPMs never miss, so performance penalties due to cache misses do not happen and the access time to the SPMs is deterministic. Third, SPMs are not coherent with the rest of memories in the system, so memory accesses to them do not generate coherence traffic. Although these advantages are very appealing to overcome the power consumption and scalability problems of muliprocessors, the main problem of SPMs is that they have to be managed by software, which degrades programmability because they require the programmer, the compiler or the runtime system to perform memory management operations.

Despite their programmability difficulties, SPMs have been successfully adopted by many architectures in numerous ways, either as the only on-chip memory or alongside a cache hierarchy. The next subsections explain how SPMs have been integrated in relevant architectures, with the architectural details of the different memory organizations and their implications on the programmability of the architecture.

2.2.1 Embedded Processors

SPMs have been widely used in embedded systems for many years. SPMs provide key advantages over caches in this domain, as they are more energy- and cost-efficient and they provide predictable access time, which is critical in hard real-time systems. For these reasons SPMs have been included in embedded processors in different ways. In processors such as the Motorola Dragonball, the Infineon XC166 or the TI TMS370CX7X SPMs are used as the only on-chip memory, while in other embedded processors like the ARM10E or the ColdFire MCF5 SPMs are introduced alongside caches.

Although in the embedded domain it is common that the programmer explicitly selects data or code to be placed in the SPMs, some works propose techniques to do it automatically in order to improve code portability. Existing approaches for automatic SPM management can be divided in two classes: static and dynamic. In static schemes designated parts of the program are allocated in the SPMs at load time, and their contents do not change during the execution of the program. In dynamic approaches parts of the program are mapped to the SPMs during the execution, dynamically changing their contents. Both categories can be further classified into techniques that consider only code, only data, or both.

Static allocation methods decide at compile time what program objects are allocated in the SPMs. The main advantage of static allocations is that they do not add any overhead at execution time. However, the amount of data or code that can be allocated in the SPMs is constrained by the size of the SPMs so, depending on the characteristics of the workload, the amount of memory accesses served by the SPMs is limited. Panda et al. [117, 118] allocate scalar constants and variables in the SPM and use heuristics to allocate those arrays that minimize cache conflicts. Angiolini et al. [5, 6] propose an algorithm based on dynamic programming to select code blocks which promise the highest energy savings. Banakar et al. [14] solve the static assignment with a knapsack algorithm for both code and data blocks. Verma et al. [151] select memory objects based on a cache conflict graph obtained through cache hit/miss statistics, selecting the optimal set of memory objects with an integer linear program variant of the knapsack algorithm. Nguyen et al. [108] propose to delay the decision of which blocks should be allocated in the SPM until the application is loaded, making it independent from the size of the SPM at the cost of embedding profiling information into the application binary.

Another approach is to dynamically map code or data to the SPMs. This method allows to map to the SPMs as much data or code as desired, but adds overheads to transfer data or code between the SPMs and the main memory during the execution of the program. Kandemir et al. [82, 81] propose to map arrays accessed in well-structured loops. Arrays are split into tiles to allow that parts of the arrays are mapped to the SPM, so arrays bigger than the SPM size can be mapped to the SPM in chunks. Li et al. [95] also focus on assigning data arrays to the SPM. To determine the most beneficial set, they first divide the SPM in partitions of different sizes and, using a conflict graph of live ranges for the arrays, a graph-coloring algorithm determines which array is mapped to which partition of the SPM at what program points. Udayakumaran and Barua [145] map data to the SPM

by constructing the control-flow graph of the program, that is annotated with timestamps to form a data program relationship graph. Using greedy heuristics, the pieces of data that maximize performance are mapped to the SPM at defined copy points in the program. Steinke et al. [137] propose to dynamically map blocks of code to the SPM before entering a loop, determining the optimal set using an integer linear program. The same technique is proposed by Egger et al. [53] but using a post-pass optimizer that operates on the binary of the program instead of the source code. Another dynamic SPM management scheme for code is proposed by Janapsatya et al. [78], where the blocks of code to be mapped to the SPM are selected based on a metric called concomitance, which indicates how correlated in time the execution of various blocks of code are. Egger et al. [54] propose to map code to the SPM using the MMU, in such a way that code is transparently mapped on demand to the SPM when a page fault happens. Cho et al. [37] apply the same MMU technique to map data to the SPM on demand.

2.2.2 Cell B.E.

The Cell B.E. [79] is one of the first and more breakthrough heterogeneous multicore processors for HPC and multimedia workloads. The architecture consists of a general-purpose core called Power Processor Element (PPE) and eight accelerator cores called Synergistic Processor Elements (SPEs) connected through a high bandwidth NoC named Element Interconnect Bus (EIB). A scheme of the architecture is shown in Figure 2.3.

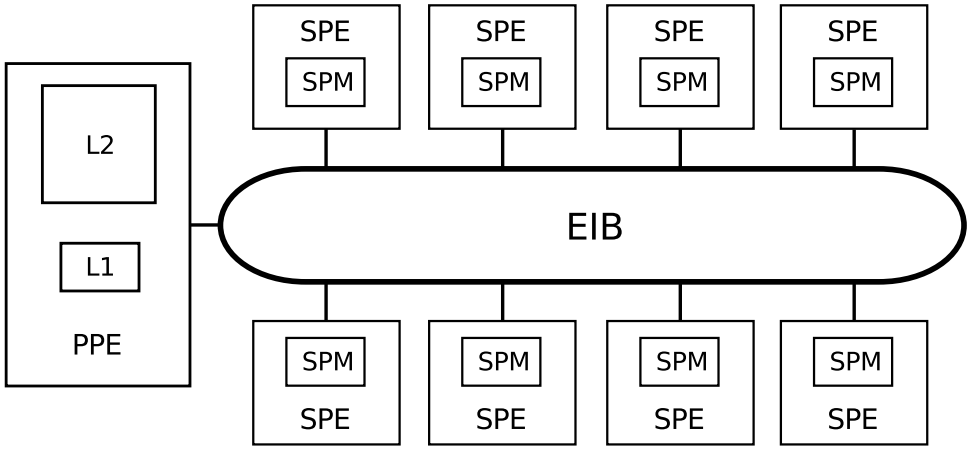

Figure 2.3: Cell B.E. architecture

The memory hierarchy of the Cell B.E. combines a hierarchy of caches and SPMs. The PPE has a private 32 KB L1 cache and a private 512 KB L2 cache, while each SPE has a private SPM of 256 KB. The SPMs of the SPEs have their own private virtual and physical address spaces and are incoherent with the rest of memories in the architecture. Each SPE can issue load and store instructions only to its private SPM and a DMA controller is used to transfer data from or to the rest of memories in the architecture. The DMA transfers are coherent with the cache hierarchy of the PPE.

Although the Cell B.E. was able to achieve significantly higher performance than the rest of processors of its time, obtaining the expected performance was complex due to the programmability difficulties the Cell B.E. suffered from. First, the programmer is in charge on distributing the computation between the PPE and the SPEs, writing the code for each element in a separate source code file. The PPE is in charge of running the OS and to start processes, which can initiate one or more SPE threads using an API to create the SPE context (`spe_context_create`), load the code and the data for the SPE context (`spe_program_load`), launch the execution of the SPE thread (`spe_context_run`), and finish the SPE thread when the execution finishes (`spe_context_destroy`). Second, the programmer has to partition the data between the tasks. The limited size of the SPMs of the SPEs makes unfeasible to statically partition the data and embed it in the SPE context so, most of the times, SPE threads use DMA transfers to dynamically transfer the data from the main memory to the SPM, perform the computation, and write the data back to the main memory if needed. An API is offered to trigger asynchronous DMA transfers (`mfc_get` from the main memory to the SPM and `mfc_put` from the SPM to the main memory) and to synchronize them (`mfc_write_tag_mask` and `mfc_read_tag_status_all`). Third, the programmer needs to perform explicit thread synchronization, and typical synchronization primitives are not supported in this architecture, so explicit messages have to be exchanged between threads. The Cell B.E. offers mailboxes to perform inter-thread communication (`spe_in_mbox_write` and `spe_in_mbox_read`), which are 32-bit messages intended to exchange program status, completion flags or memory addresses.

Several approaches have been proposed to support shared memory programming models on the Cell B.E., aiming to overcome its programmability issues. The IBM XL compiler for the Cell B.E. supports OpenMP constructs [112] in two ways. In both approaches the idea is to use a software cache on each SPE to automatically transfer data between its SPM and the main memory. The first technique [55, 56] introduces a software cache

that surrounds every memory reference with a piece of code that mimics the operation of a cache: lookup of the address, placement, replacement of an old entry, data transfer and synchronization. Although some optimizations such as using asynchronous data transfers [12] can improve the initial design, the amount of code executed prior to every memory access imposes severe performance overheads. The second technique [36, 67] drastically reduces the overheads for strided memory references by applying loop tiling, in such a way that strided memory references are handled in the outermost loop by a software cache that tracks data at the granularity of big blocks, while random memory references are handled by a second software cache at every access in the innermost loop.

COMIC [92] is a low level shared memory interface for the Cell B.E. that can be used directly by the programmer or as a target of compilers for shared memory programming languages. COMIC uses a centralized coherence protocol on the PPE and wraps all memory references of the SPE code with a software cache that requests memory pages if needed, and proposes optimizations to reduce communication for read-only and single writer pages. Although these optimizations enable important performance improvements over the techniques introduced in the IBM XL compiler, applying them at compile time requires the compiler to ensure that there are no memory aliasing hazards between the memory references of the code. This can be done in simple codes but, in the presence of pointers and functions, it becomes a very challenging problem for the compiler.

As an alternative, other works propose to program the Cell B.E. using task data-flow programming models. An initial version of OmpSs for the Cell B.E., called CellSs [17, 120], allows programmers to annotate functions as tasks and to specify their dependences, and a source to source compiler uses this information to generate code for the PPE and for the SPEs. Similarly, Sequoia allows to model the memory hierarchy of the Cell B.E. and to decompose the program in a hierarchy of tasks that are executed on the SPEs. In both approaches a runtime system manages the execution of the tasks in a data-flow fashion and orchestrates the DMA transfers for the task dependences, transferring the input dependences of a task to the SPM of the SPE that is going to execute it and writing back the output dependences to main memory when the task finishes. The runtime system of the CellSs also uses a locality aware scheduling policy to reduce data motion and applies renaming on the output dependences to remove Write after Read (WaR) and Write after Write (WaW) dependences between tasks. Further optimizations for the CellSs include SPE-to-SPE DMA transfers [18], lazy write back of output dependences [19] or distributed scheduling among SPEs [20].

2.2.3 GPGPUs

General-Purpose Graphics Processing Units (GPGPUs) are accelerators designed to execute massively parallel workloads very efficiently. These architectures have received a lot of attention in the HPC domain during the last years, specially the ones manufactured by NVIDIA.

In HPC systems GPGPUs are attached to a general-purpose processor (called the host processor), and they have different address spaces. The host processor is in charge of running the OS, starting applications and executing the sequential parts of the applications. The GPGPU executes parts of the applications that expose a high degree of parallelism, which are encapsulated in kernels. When the application encounters a kernel, the host processor transfers the code of the kernel and the data accessed by the kernel to the GPGPU, triggers its execution, and transfers the data back to the main memory if needed.

A scheme of the architecture of a NVIDIA GPGPU is shown in Figure 2.4. The architecture consists of many cores (called Streaming Multiprocessors or SMs in NVIDIA's nomenclature) that execute groups of threads in lock-step. The cores have the ability to perform very fast context switches between groups of threads so that the memory latency is not a limiting factor for performance. The memory hierarchy of the GPGPU consists of a main memory and a L2 cache shared between all the SMs. Each SM has a SPM, called shared memory, and two first level caches: the L1 cache and a read-only texture cache. The L1 cache, the texture cache and the SPM of every SM are incoherent with the rest of memories in the system. In order to manage this complex memory hierarchy the main memory is divided in five memory spaces, that operate as follows:

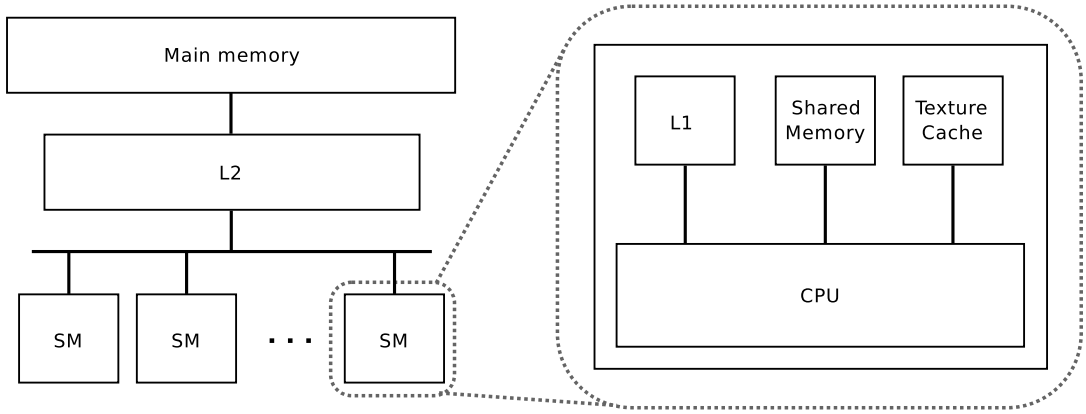

Figure 2.4: GPGPU architecture

- The global memory is a memory space where the programmer can explicitly transfer data. When a SM issues a memory request to the global memory the data is cached in the shared L2 cache and either in the L1 or in the texture cache of the SM. If the request is read-only it is cached in the texture cache, otherwise it is cached in the L1 cache.

- The constant memory is a read-only memory space where the programmer can explicitly allocate data. When a SM issues a memory request to the constant memory the data is cached in the shared L2 cache and in the texture cache of the SM.

- The texture memory is a read-only memory space where the programmer can explicitly allocate data. When a SM issues a memory request to the texture memory the data is cached in the shared L2 cache and in the texture cache of the SM.

- The shared memory is a memory space where the programmer can explicitly allocate variables that are shared between the threads. The compiler allocates the necessary space in the SPM of the SM for the variable, and memory requests from a SM to the shared memory only access its SPM.

- The local memory is a memory space where the compiler can reserve space for register spilling. When a SM issues a memory request to the local memory the data is cached in the shared L2 cache and in the L1 cache of the SM.

The most common way to program heterogeneous systems with GPGPUs in HPC is using CUDA [110]. In this programming language the programmer has to distribute the computation between the host processor and the GPGPU, synchronize the execution between the two processing elements, and explicitly manage the memory hierarchy of the accelerator. CUDA provides language extensions to C, C++ and Fortran to perform these actions. In order to specify kernels for the GPGPU CUDA provides the `__global__` function specifier, and kernels are launched with a special syntax <<< ... >>> to specify the number of threads that execute the kernel. In addition, the code of the kernels is not written in traditional imperative form. Instead, the code of the kernel only specifies the computation that has to be performed by one thread, and this code is replicated among all the threads in a transparent way, so the parallelism is implicit. For synchronization, CUDA offers the function `__syncthreads` to synchronize the threads within a kernel, and methods such as `cudaDeviceSynchronize` to synchronize the execution of the kernels with the host code. CUDA also offers functions and variable specifiers

to manage the memory hierarchy of the GPGPU. In order to transfer data to the global memory the programmer has to allocate space using `cudaMalloc`, transfer the data using `cudaMemcpy` and free the space using `cudaFree`. CUDA also offers the variable specifiers `__constant` and `__shared` to statically allocate data to the constant memory and to the shared memory, respectively. To allocate data in the texture memory the programmer has to define a texture handler that specifies the memory layout of the data, while the local memory is transparently managed by the compiler. In addition, in order to decide if the memory accesses to the global memory are cached in the L1 cache or in the texture cache, the compiler applies alias analyses between the memory references of the code to guess if the data being accessed is read-only, and generates a different type of instruction (`LDG`) for memory accesses to read-only data. In non-trivial cases the compiler does not succeed in ensuring the data is read-only so the programmer has to use the `__ldg` intrinsic in the memory references to the global memory to enable the usage of the texture cache for these memory accesses.

Many works aim to ease the programmability of GPGPUs. One approach is to program the GPGPUs with CUDA and alleviate the burden of explicitly transfer data between the host processor and the accelerator, either allowing the compiler to generate code for the data transfers [77] or transparently moving the data during the execution [62, 63]. Another approach is to use extensions on existing programming languages to program the GPGPUs, allowing the compiler of the respective programming languages to generate code for the data transfers. This kind of language extensions have been proposed for Java [50], Unified Parallel C [35], Global Arrays [143], Chapel [135], Fortran [155] and C++ [68]. Offload programming models like OpenHMPP [49] and OpenACC [113] take a similar approach, but using compiler directives instead of language extensions. In addition, GPGPUs are also supported in languages such as OpenCL [138] or Kokkos [52], which are not simpler than CUDA but they offer generic abstractions to program accelerators in a portable way.

Another way to solve the programmability issues of GPGPUs is to use shared memory programming models. OpenMPC [93, 94] does source to source transformations of OpenMP loops to CUDA kernel functions and adds code to transfer data between the host processor and the GPGPU. Moreover, some task data-flow programming models such as OmpSs [121], StarPU [8], Sequoia [58], Legion [16] or Habanero [134] are able to manage GPGPUs, giving the runtime system the responsibility of scheduling tasks to the GPGPUs and triggering the data transfers specified in the data dependences of the tasks.

All these approaches automatically perform the memory management operations required by the GPGPUs or make it easier for the programmer to program them. However, many of them make a suboptimal use of the memory hierarchy of the GPGPU, as they only use the global memory. None of the solutions that automatically manage the data are able to allocate it in the constant memory, the texture cache, or the shared memory. Similarly, most of the solutions that extend existing programming languages do not allow to specify data for these three memory spaces. As a consequence, the advantages provided by the shared memory and the texture cache of the GPGPU are not exploited in many of these solutions.

2.2.4 Stream Architectures

Some stream architectures and stream extensions for generic architectures also use SPMs in their memory hierarchy.

The Imagine [85] is a stream accelerator for media processing. Its architecture consists of a microcontroller that stores VLIW instructions, eight arithmetic clusters with eight functional units each, a local register file, and a 128 KB Stream Register File (SRF), which is an array of memory that works as an SPM. The Merrimac [42] is a stream accelerator for HPC with a very similar architecture. It has twice the number of arithmetic clusters with a different mix of functional units that are better suited for HPC workloads, a correspondingly larger SRF of 1 MB, and hardware support for gather-scatter memory instructions. In order to operate these accelerators the host processor is in charge of dividing the size of the SRF in streams, loading data for the input streams from the main memory to the SRF, loading the code in the accelerator, triggering the execution, and storing the data of the output streams from the SRF to the main memory. When the host processor triggers the execution the accelerator performs compound stream operations on every element of the input streams. Every compound stream operation reads an element from the input streams in the SRF, computes the arithmetic operations specified in the code storing the temporary results in the local register file, and writes the outputs to the output streams in the SRF.

The Imagine and the Merrimac are programmed with the StreamC and the KernelC programming languages. These are programming languages that offer support for streams with a syntax similar to C. StreamC is the language for the host processor side, that provides operations to load streams from the main memory to the SFR, store streams from the SFR to the main memory, load the code in the accelerator, and trigger the execution.

KernelC is used to program the kernels for the accelerator, and provides special constructs to read and write elements from the streams to the local register file. In these programming models the scheduling of kernels and stream memory operations is statically generated at compile time [43]. The compiler analyzes the structure of the program to estimate the time spent in kernels and memory operations and generates an schedule that overlaps the kernel execution with the memory transfers. Moreover, the compiler applies optimizations such as loop tiling, loop unrolling or software pipelining at the level of kernels and stream memory operations to optimize the usage of the SRF.

Gummaraju et al. [70, 69]. propose to adopt the stream execution model in general-purpose processors. In their initial proposal [70] they avoid any change in the architecture. Instead, they use cache locking on the L2 cache to emulate the SRF and two separate threads to emulate the data transfer engine and the computational engine of stream architectures. SMT is exploited to run the two threads concurrently on the same core, allowing to overlap the data transfers to the SRF with the execution of computational kernels. In their following work [69] the authors propose hardware modifications to general-purpose processors to better support the streaming model, extending the prefetcher of the L2 cache with a programmable engine that performs asynchronous bulk memory transfers between the SRF and the main memory and extra logic to pack data in vectors for the SIMD units.

2.2.5 Other Architectures

The SARC architecture [125] is a heterogeneous architecture with clusters of master and worker cores. Similar to the Cell B.E., the master cores are general-purpose cores with private L1 caches that are responsible for running the OS, starting applications and spawning tasks to the worker cores. The worker cores in charge of running the tasks have different ISAs and pipelines optimized for different HPC computations, and all of them are equipped with a virtually-indexed virtually-tagged L0 cache, a private L1 cache, and a SPM. The architecture also has a shared distributed L2 NUCA cache that serves misses from the L1 caches of all cores and DMA transfers for the SPMs. The L2 cache is inclusive of all L1 caches and, in order to maintain coherence between the L2 and the L1 caches, the cache directory keeps per-cluster presence bits instead of per-core presence bits, and invalidations are broadcast inside each cluster. The SARC architecture is intended to be programmed with task data-flow programming models, where the runtime system is in charge of scheduling tasks to the most appropriate worker cores and of transferring the data specified in the task dependences to the SPMs of the worker cores.

The Runnemede [33] is a modular and hierarchical architecture for HPC. The basic module of the Runnemede is called block, which consists of a general-purpose core, eight execution engines, an intra-block network and a L2 SPM of 2.5 MB. The general-purpose cores have a private 32 KB L1 cache and a pipeline optimized for latency-sensitive OS operations and sequential program phases, while the execution engines are processors optimized for throughput with a private 32 KB L1 cache and a 64 KB L1 SPM. Blocks are organized in units, which consist of 8 blocks, a L3 SPM of 8 MB and an intra-unit network. A Runnamede chip contains 8 units and a 16 MB L4 SPM, all connected through another network. The Runnemede is programmed with the Codelets task data-flow programming model, giving the runtime system the responsibility of scheduling tasks to the execution engines and triggering the necessary data transfers between the SPMs of the system. In this hierarchical architecture it is very important that the task scheduling is performed in such a way that data locality is maximized.

The polymorphus TRIPS architecture [128, 129] consists of four partitionable large cores and a reconfigurable memory hierarchy. The cores are a group of execution nodes, which contain a bank of the L1 instruction cache, a bank of the register file, a set of reservation stations, an integer ALU, a floating point unit, and a bank of the L1 data cache. This core organization allows to partition them to exploit different forms of parallelism. The cores are connected to an array of 32 KB memory tiles that can be configured to behave as L2 cache banks or SPMs. When configured as SPMs the tag array of the tiles is turned off, and the cache controllers are augmented with DMA-like capabilities to transfer data between the tiles and the main memory. Under this configuration the TRIPS architecture is programmed with stream programming models like StreamC and KernelC to manage the SPMs.

The Smart Memories Architecture [98, 99] is a tiled manycore with a highly configurable memory hierarchy. Each tile of the processor contains 16 blocks of 8 KB SRAMs, and each memory block is equipped with row and column decoders and a write buffer. Memory blocks can act independently as a SPM or they can be grouped to act as a cache, devoting some blocks to the tag array and some others to the data array. These features allow the memory hierarchy of the Smart Memories Architecture to be configured as a set of SPMs, a set of caches, or a combination of both. The paper focuses on the hardware details that allow the reconfiguration, but does not specify how the resulting system would be exposed to the software layers nor how the memory hierarchy would be managed when it is configured as SPMs.

Cook et al. [38] propose to use cache locking to configure a part of the L1 cache of every core as a Virtual Local Store (VLS). A portion of the virtual address space is direct-mapped to a range of the physical address space to access the VLSs and, in order to overcome some of the drawbacks of the locking mechanism, the authors propose hardware modifications that allow the accesses to the virtual address range of the VLSs to be done without accessing the TLB nor the tag array of the cache. The architecture also includes a DMA controller in every core that moves data between the VLS and the main memory, and an API is provided to the software to trigger the data transfers. In this work it is the programmer who explicitly writes the code to manage the VLSs, applying tiling transformations to the loops to trigger the DMA transfers in the outermost loop.

Bertran et al. [22] propose to add a SPM alongside the L1 cache of a single core general-purpose processor. The SPM is exposed to the software using a reserved range in the virtual and physical address spaces and memory accesses to the SPM bypass the TLB by performing a range lookup with a set of registers that keep the virtual-to-physical address mapping. A DMA controller is also introduced in the core to transfer data between the SPM and the main memory. In order to manage the SPM a tiling software cache [67] is used to map strided memory references, while random memory references access the cache hierarchy. The authors give the compiler the responsibility of doing the code transformations for the software cache, but they do not solve the memory aliasing problem between memory references to the SPM and to the cache hierarchy. In their design, if the compiler cannot ensure there is no aliasing between strided and random memory references, the code transformations for the software cache are discarded and the SPM is not used. This problem greatly restricts the effective utilization of the SPM in non-trivial codes.

2.2.6 Summary

SPMs have been adopted in many commercial processors and research projects in the fields of embedded computing and HPC, where power efficiency and scalability are very important factors for the success of the architectures. Although memory hierarchies with SPMs offer important advantages over cache-coherent shared memory organizations, the programmability issues they impose are a major concern in current solutions.

The fact that the most relevant architectures with SPMs are coupled to ad-hoc programming models and language extensions clearly shows that the programmability burdens introduced by the SPMs still need to be solved. Many proposals attempt to auto-

matically manage the SPMs or to support shared memory programming models in these architectures, but they all have severe limitations or are only applicable to the specific architecture they target.

In order to make a step forward in this direction, this first contribution of this thesis proposes a solution for the memory aliasing problem encountered by compiler techniques that generate code to manage the SPMs with tiling software caches from fork-join programming models. The second proposal of this thesis shows that task data-flow programming models are also very suitable for a memory organization that combines caches and SPMs, preserving the goodnesses of cache-coherent shared memory hierarchies while exploiting the benefits provided by the SPMs.

Chapter 3
Methodology

This chapter describes the experimental methodology followed in this thesis. The first section explains the simulation infrastructure, focusing on the simulators used to evaluate the proposals, the baseline architecture that is modeled in the simulators, and the software environments for the fork-join and the task programming models that run on top of the architecture. The second section discusses the benchmarks for both programming models, with their main characteristics and the setup employed in the experiments. Finally, the third section defines the metrics used to evaluate the proposals of this thesis.

3.1 Simulation Infrastructure

The simulation infrastructure used in this thesis consists of several pieces. Two simulators are set up to evaluate the performance and the energy consumption of the baseline architecture and the proposals for fork-join and task programming models. Two runtime systems run on top of the simulated architecture, one for each programming model, that manage the execution of the parallel benchmarks used in the experiments.

3.1.1 Simulators

The baseline architecture and the hardware extensions proposed in this thesis are modeled with Gem5 [25]. Gem5 is an execution-driven cycle-accurate full system multicore simulator that supports various ISAs and includes multiple models for the CPU and for the memory hierarchy, with different levels of detail. The x86 cycle-accurate detailed out-of-order core model and the detailed memory hierarchy model are used in this thesis.

The energy consumption and the area of the architecture is modeled with McPAT [97]. McPAT is an integrated power, area, and timing simulator for multicore architectures built

on top of CACTI [141, 142]. It includes high performance and low power models for in-order and out-of-order cores, NoCs, caches, and memory controllers, and has support for multiple process technologies and clock gating schemes. In this thesis it is used a process technology of 22 nm with the default clock gating scheme of the tool, and all the components of the architecture are modeled with low power designs except those that do not fit in the cycle time, for which high performance designs are used.

3.1.2 Baseline Architecture

The baseline architecture of the thesis, shown in Figure 3.1, is a multiprocessor with a hybrid memory hierarchy. The hybrid memory hierarchy assumed in this thesis consists of extending every core of the shared memory multiprocessor with a SPM and a DMA controller (DMAC).

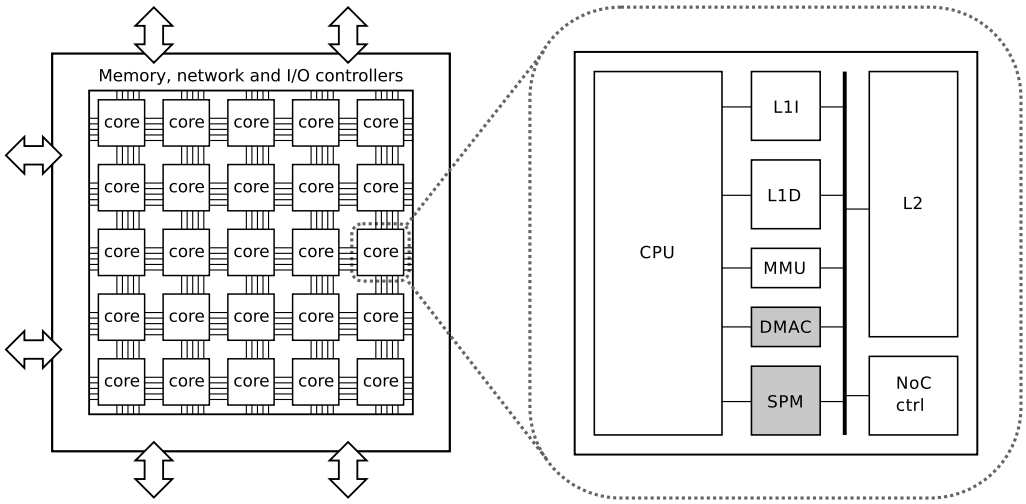

Figure 3.1: Baseline architecture

The SPMs are added alongside the L1 cache of every core, and they are accessible by all the cores. The system reserves a range of the virtual and physical address spaces for each SPM of the chip, and direct-maps the virtual ranges to the physical ones, as shown in Figure 3.2. Every core keeps the address space mapping in eight registers, four to store the starting and the final virtual addresses of the local SPM and of the global range of the SPMs, and four to keep the physical address space of all the SPMs and of the local SPM. These registers are used to identify virtual addresses that access the SPMs and to do the virtual-to-physical address translation, allowing all the cores to access any SPM by issuing loads and stores to their virtual address ranges. At every memory instruction, before any

MMU action takes place, a range check is performed on the virtual address. If the virtual address is in the range of some SPM, the MMU is bypassed and a physical address that points to the SPM is generated. Apart from the simplicity of the implementation, an important advantage of this way of integrating the SPMs [22, 38] is that no pagination is used, so memory accesses to them do not need to lookup the TLB, minimizing the energy consumption and ensuring deterministic latency. In addition, the typical size of SPMs is orders of magnitude smaller than the size of the RAM and the virtual address space of a 64-bit processor, so the virtual and physical address ranges reserved for the SPMs occupy a very minor portion of the whole address spaces.

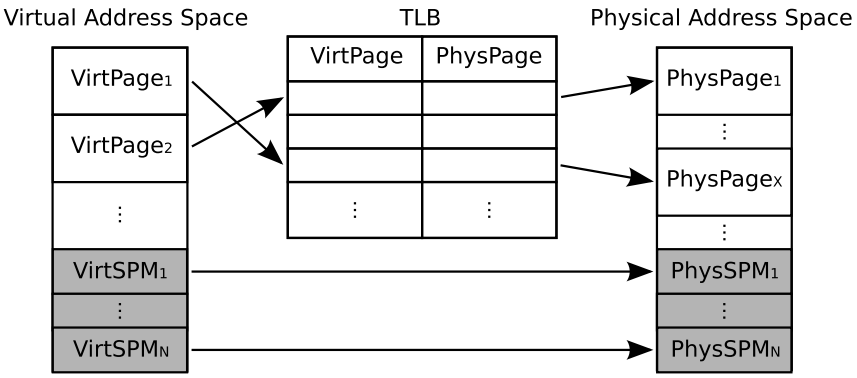

Figure 3.2: Address space mapping for the SPMs

The DMA controllers transfer data between the SPMs and the global memory (GM, which includes caches and main memory). They support three operations: (1) *dma-get* transfers data from the GM to a SPM, (2) *dma-put* transfers data from a SPM to the GM and (3) *dma-synch* waits for the completion of certain DMA transfers. The *dma-get* and *dma-put* operations support block transfers and scatter-gather transfers with simple strides [69]. Every DMA controller exposes a set of memory-mapped I/O registers to the software so it can explicitly trigger the DMA operations. DMA transfers are integrated in the cache coherence protocol of the GM [21, 88]. The bus requests generated by a *dma-get* look for the data in the caches and read the value from there if it exists, otherwise they read it from the main memory. The bus requests of a *dma-put* copy the data from the SPM to the main memory and invalidate the cache line in the whole cache hierarchy.

The main parameters of the processor configuration are summarized in Table 3.1. The number of cores and the variations of some parameters are specified in the evaluation of each proposal.

Table 3.1: Processor configuration

Cores	Out-of-order, 6 instructions wide, 2 GHz
Pipeline front end	13 cycles. Branch predictor 4K selector, 4K G-share, 4K Bimodal. 4-way BTB 4K entries. RAS 32 entries
Execution	ROB 160 entries. IQ 64 entries. LQ/SQ 48/32 entries. 3 INT ALU, 3 FP ALU, 3 Ld/St units. 256/256 INT/FP RegFile. Full bypass.
L1 I-cache	2 cycles, 32 KB, 4-way, pseudoLRU
L1 D-cache	2 cycles, 32 KB, 4-way, pseudoLRU, stride prefetcher
SPM	2 cycles, 32 KB, 64 B blocks
DMA controller	DMA command queue 32 entries, in-order Bus request queue 512 entries, in-order
L2 cache	Shared unified NUCA sliced 256 KB / core 15 cycles, 16-way, pseudoLRU
Cache coherence	Real MOESI with blocking states, 64 B block size distributed 4-way cache directory 64K entries
NoC	Mesh, link 1 cycle, router 1 cycle

3.1.3 Operating System Support

The hybrid memory hierarchy requires some changes at the OS level to manage the SPMs.

The contents of the SPMs and the address space mapping registers need to be saved and restored at context switches. For this purpose, the OS process structure is extended with the fields necessary to store the address mapping registers for the SPMs. By default, processes execute with this mapping disabled, so the SPMs are not accessible. Whenever a SPM-enabled application starts, the OS configures the registers for the address spaces of the SPMs and stores their values in the process structure. Whenever a process is scheduled, the registers are set to the values stored in the process structure. In addition, the OS must also save and restore the contents of the SPMs whenever a process is scheduled for execution. In order to keep these overheads low, SPM contents can be switched lazily, similar to how the Linux kernel does for the floating point register file [76].

Since multiple concurrent applications can be using the SPMs, a register that contains a single bit for each SPM in the system is added to each core. The Nth bit of this register identifies whether the Nth SPM can be accessed, and accessing a SPM without that bit set raises an exception. This mechanism allows to control which SPMs are available to the running processes, enabling that multiple concurrent processes can safely use the SPMs.

These OS modifications provide backwards compatibility and can also be used by the OS to improve energy consumption by powering down the SPMs that are not being actively used. As a result, the only overhead when running in compatibility mode is the unused area of the SPMs and the DMA controllers.

3.1.4 Runtime Systems

Two runtime systems are used to manage the parallel execution in fork-join and in task programming models.

The runtime system of the fork-join programming model is a library that automatically does a static distribution of the iteration space of a loop between threads and manages the SPMs of the hybrid memory hierarchy with a tiling software cache. The initial implementation of the runtime library [67] is designed for the Cell B.E. processor, which internally contains two software caches for every core, one for strided memory accesses and another one for random memory accesses. The runtime library used in this thesis is a modification of the initial implementation, adapted to work on the baseline architecture with the hybrid memory hierarchy. First, the software caches for random memory accesses are removed from the library, because in the hybrid memory hierarchy these are handled by the cache hierarchy. The logic of the tiling software caches for the strided memory accesses is left untouched, but the back-end functions are reimplemented to interact with the simulator. The address spaces of the SPMs of the cores are defined by the runtime library as memory regions in the address space of the application, and these memory regions are registered in the simulator at the beginning of the execution. At simulation time, any load or store to these address ranges is served by the simulated SPMs. The interfaces for the DMA controllers are also redefined. The original routines to trigger the DMA transfers are reimplemented so that they perform stores to the addresses of the memory-mapped I/O registers exposed by the simulated DMA controllers. At simulation time, the memory operations to these addresses are captured and sent to the simulated DMA controllers, which form the corresponding DMA commands. Similarly, the original routine to wait for the completion of certain DMA transfers is reimplemented to do a polling loop that performs loads to the addresses of the memory-mapped I/O registers where the simulated DMA controllers expose the DMA completion flags to the software.

For the task programming model this thesis uses Nanos++ [51] version 0.7a as runtime system. Nanos++ natively supports the OpenMP 4.0 task directives and the additional tasks constructs provided by OmpSs. For this thesis Nanos++ is configured to work on a shared memory multiprocessor and is extended with the mechanisms proposed in Chapter 5 to manage the SPMs of the hybrid memory hierarchy. In the simulated environment the runtime system runs on top of a Gentoo Linux with a kernel 2.6.28-4.

3.2 Benchmarks

The NAS Parallel Benchmarks (NPB) [10] are used for the evaluation of the proposal on automatic management of SPMs in fork-join programming models. The NPB suite is a set of benchmarks designed to evaluate the performance of parallel supercomputers. The benchmarks are derived from typical HPC applications and consist of several kernels and pseudoapplications implemented in commonly used programming models. Six benchmarks extracted from the OpenMP implementation of the version 3.0 of the benchmark suite are used in this thesis. The benchmarks are compiled using GCC 4.7.3 with the -O3 optimization flag on. Table 3.2 shows the selected benchmarks, their descriptions and the input sets used in the experiments.

Table 3.2: Fork-join benchmarks

Benchmark	Description	Input
CG	Estimates the smallest eigenvalue of a sparse symmetric positive-definite matrix using the inverse iteration with the conjugate gradient method as a subroutine for solving systems of linear equations	Class B
EP	Generates independent Gaussian random variates using the Marsaglia polar method	Class A
FT	Solves a three-dimensional partial differential equation using the fast Fourier transform method	Class A
IS	Sorts a large amount of small integers using the bucket sort algorithm	Class A
MG	Approximates the solution of a three-dimensional discrete Poisson equation using the V-cycle multigrid method	Class A
SP	Solves a synthetic system of nonlinear partial differential equations using a scalar pentadiagonal solver method	Class A

The evaluation of the proposal on automatic management of SPMs in task programming models uses two parallel applications from the PARSEC suite [24] and a set of representative HPC numerical kernels. The benchmarks are compiled with the Mercurium 1.99.0 [11] source to source compiler, which translates the original C/C++ code of the benchmarks with pragmas for the task annotations to a C/C++ code with function calls to Nanos++, which is the runtime system used for the task programming model. The resulting code is compiled with GCC 4.7.3 using the -O3 optimization flag to generate the final binary. Table 3.3 shows the benchmarks used, their descriptions and the inputs used in the evaluation of the proposal.

Table 3.3: Task benchmarks

Benchmark	Description	Input
blackscholes	Calculates the prices for a portfolio of European options using the Black-Scholes partial differential equation	PARSEC Simlarge
fluidanimate	Simulates the interaction of particles in an incompressible fluid using the Smoothed Particle Hydrodynamics method	PARSEC Simlarge
jacobi	Solves a diagonally dominant system of linear equations using the Jacobi method	2560x2560 floats
kmeans	Partitions elements into clusters in which each element belongs to the cluster with the nearest mean	200K elements 15 clusters
knn	Classifies elements by a majority vote of its neighbors, assigning the elements to the most voted class among its K nearest neighbors	4096 elements 20 neighbors 15 classes
md5	Produces a 128-bit hash value using a cryptographic hash function	6400 32 KB buffers
raytrace	Generates an image by tracing the path of light through the pixels and simulating the effects of its encounters with objects, and then rotates the resulting image	1024x1024 image 3 ray shots per pixel rotation of 3°
tinyjpeg	Decodes JPEG images with fixed encoding of 2x2 MCU size and YUV color	JPEG image of 2992x2000 pixels
vecadd	Calculates the sum of two vectors	1024M integers
vecreduc	Calculates the sum of all the elements of a vector	1024M integers

3.3 Metrics

The outcome of the evaluation of the proposals of this thesis proves that multiprocessors with hybrid memory hierarchies provide several benefits when compared to shared memory multiprocessors. Furthermore, it proves that the ideas proposed in this thesis solve the programmability issues of hybrid memory hierarchies with negligible or very affordable overheads. In order to do these demonstrations the evaluation of this thesis is based on three metrics: performance, NoC traffic and energy consumption.

The performance numbers are shown in cycles, since nor the hybrid memory hierarchy nor any of the proposals have any impact on the cycle time. The evaluation section of each proposal provides deeper analyses to explain the performance differences, using lower level metrics such as the execution time overhead imposed by managing the SPMs in software, the hit ratio of the caches, the congestion in the NoC or the overheads added by the mechanisms proposed in this thesis.

The NoC traffic is another important metric to be studied, since the introduction of the SPMs and the DMA controllers drastically changes the way the data is moved inside the chip. This thesis studies the impact of the hybrid memory hierarchy in NoC traffic, categorizing the packets generated for fetching instructions, reading data, writing data,

moving data using DMA transfers, and performing cache management actions such as block write-backs and replacements.

Another of the goals of the introduction of the SPMs is to reduce the energy consumption. For this reason, the evaluation of this thesis reports the Joules consumed by a multiprocessor with a hybrid memory hierarchy and with a cache hierarchy, and also shows the overheads introduced by the different proposals in terms of energy consumption. Detailed analyses show how the energy consumption is distributed between components, specially focusing on the CPU, the cache hierarchy, the SPMs, and the NoC, and the causes of the variations in the consumed energy are also explained.

These are the three main metrics used to evaluate the proposals of this thesis. Detailed analyses are provided to explain the results shown by these three metrics, taking into consideration the additional low level metrics that are important in each particular study.

Chapter 4

SPM Management in Fork-Join Models

Fork-join programming models are the most commonly used way to program shared memory multiprocessors. The main reason for the success of this approach is its programming simplicity, that is granted by the ability of cache-coherent memory hierarchies to manage the data transparently to the software. Unfortunately, the introduction of SPMs in the memory hierarchy imposes a much more complex memory model where data must be managed by software. As a consequence, multiprocessors with SPMs are usually programmed with programming models that require the programmer to explicitly manage the data. In order to alleviate this programming burden, a promising solution is to give the compiler the responsibility of automatically generating code to manage the SPMs from fork-join programming models, transparently to the programmer.

Tiling software caches [55, 56, 67, 133] are well-known technique to automatically manage SPMs. Tiling software caches manage the SPMs by dynamically mapping big chunks of data accessed by strided memory accesses, taking advantage of their high spatial locality and predictability. Random accesses, on the other hand, are difficult to manage by software because of their unpredictability, so they greatly benefit from the ability of caches to automatically request data and to keep it coherent. This approach is very suitable for HPC applications, which are dominated by strided accesses to big input sets [107, 152], so the amount of data mapped to the SPMs and the number of memory accesses served by them are maximized.

Compilers succeed in generating code to manage the SPMs with tiling software caches when the computation is based on predictable memory access patterns but, when unpredictable memory access patterns are found, they encounter important limitations [55, 56, 67]. Due to the incoherence between the SPMs and the cache hierarchy, the compiler cannot generate code to manage the SPMs if it cannot ensure that there is no aliasing between two memory references that may target copies of the same data in the SPMs and

in the cache hierarchy. This memory aliasing problem greatly restricts the ability of the compiler to generate code for hybrid memory hierarchies in non-trivial cases.

This chapter proposes a coherence protocol for hybrid memory hierarchies that allows the compiler to always generate code to manage the SPMs with tiling software caches, even in the presence of memory aliasing hazards. In a hardware/software co-designed mechanism, the compiler identifies memory accesses that may access incoherent copies of data, and the hardware diverts these accesses to the memory that keeps the valid copy of the data. The proposal allows the compiler to always generate correct code to manage the SPMs, so architectues with hybrid memory hierarchies can be exposed to the programmer as a shared memory multiprocessor, maintaining the programming simplicity of fork-join programming models and preserving code portability.

4.1 Tiling Software Caches

Tiling software caches [55, 56, 67, 133] are a well-known approach to automatically manage SPMs without any intervention from the programmer. In order to enable this technique, the compiler applies a series of analyses and code transformations on the loops of a code written in a fork-join programming model.

The first step is to identify data suitable to be mapped to the SPMs. In fork-join programming models the compiler uses analyses and code annotations provided by the programmer to decide how the data and the computation is distributed among the threads. Based on this distribution, it identifies array sections [116] that are sequentially traversed and private to each thread. These array sections are the preferred candidates to be mapped to the SPMs because the strided accesses used to traverse them are highly predictable and their privateness avoids costly data synchronization mechanisms on the SPMs.

After identifying the array sections to be mapped to the SPMs the compiler does the code transformations, inserting calls to a runtime system that will manage the DMA transfers at execution time. For a computational loop the code is transformed into a two-nested loop that uses tiling to do the computation [55, 56, 67, 133]. As shown in Figure 4.1, each iteration of the outermost loop executes three phases: (1) a control phase that maps chunks of the array sections to the SPMs, (2) a synchronization phase that waits for the completion of the DMA transfers, and (3) a work phase that performs the computation for the currently mapped chunks of data. These phases repeat until the whole iteration space is computed.

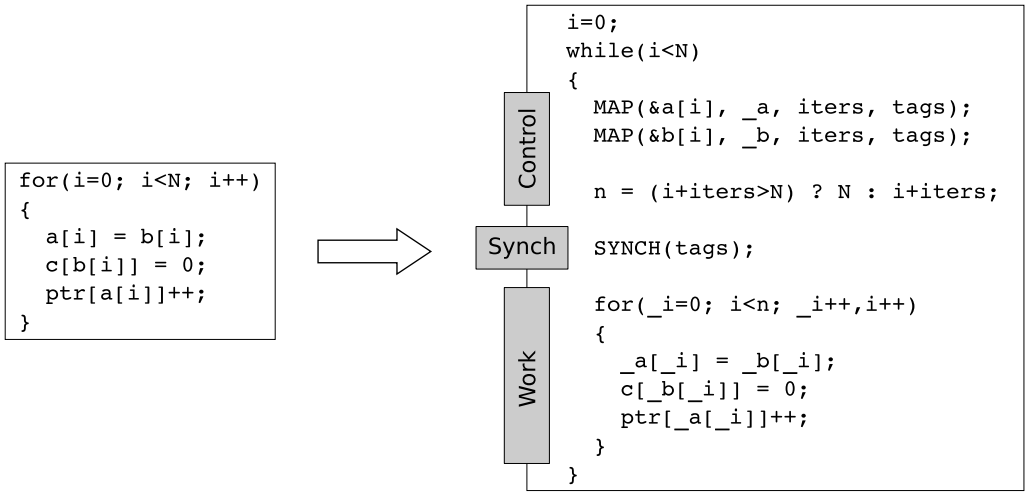

Figure 4.1: Code transformation for tiling software caches

Before entering the loop, the runtime divides the size of the SPM in equally-sized buffers in order to minimize complexity and overheads. One buffer is allocated for each memory reference that is mapped to the SPM. In Figure 4.1, `ALLOCATE_BUFFERS` allocates two buffers to map chunks of `a` and `b`, and each buffer occupies half the size of the SPM.

The control phase moves chunks of array sections between the SPM and the GM. At every instance of the control phase, in the `MAP` statement for each array section, the chunk of data for the next work phase is mapped to its corresponding SPM buffer with a *dma-get*, and the previously used chunk is written back to the GM if needed with a *dma-put*. Each call to `MAP` also sets a pointer to the first element of the buffer that has to be computed (`_a` and `_b`), updates the number of iterations that can be performed with the current mappings (`iters`) and sets the tags associated to the DMA transfers (`tags`).

After waiting for the DMA transfers to finish in the synchronization phase, the work phase takes place. This phase does the same computation as the original loop, but with two differences. First, the memory references to the array sections (`a` and `b`) are substituted with their corresponding SPM mappings (`_a` and `_b`). Second, the iteration space of the work phase is limited to the number of iterations that can be performed with the chunks of data currently mapped to the SPM.

4.1.1 Coherence Problem

The hybrid memory hierarchy opens the door to incoherences between copies of data in the SPMs and the GM. When a chunk of data is mapped to some SPM, a copy of the data is created in its address space, and the coherence between the copy in the SPM and the copy in the GM has to be explicitly maintained because there is no hardware coherence between the two memory spaces. This issue restricts the compiler from performing the code transformation for tiling software caches in non-trivial cases.

Once the compiler has identified the array sections to be mapped to the SPMs, it changes the memory references in the work phase so that they access the copy in the SPMs, while the rest of memory references access the GM. This causes that two incoherent copies of the same data can be accessed simultaneously during the computation, one via strided accesses to the SPMs and the other via random accesses to the GM, resulting in an incorrect execution. In order to ensure the correctness of the code transformation, the compiler has to apply alias analyses [45, 46, 73, 74, 90, 91, 153, 154] between the memory references that target the SPMs and the rest of memory references in the loop body and ensure there is no aliasing. In the example in Figure 4.1 this implies predicting if any instance of the accesses to `c` or `ptr` aliases with any instance of the accesses to `a` or `b`. This problem, that also affects compiler auto-vectorization and auto-parallelization [73, 74], has not been solved in the general case, especially in the presence of pointers.

This chapter proposes an efficient mechanism that ensures coherence in hybrid memory hierarchies. The solution avoids the limitations stemming from the inability to solve the memory aliasing problem, bringing the optimization opportunities to a new level where compilers no longer have to discard the code transformations for tiling software caches due to memory aliasing hazards.

4.2 Coherence Protocol

The main idea of the coherence protocol is to avoid maintaining two coherent copies of the data but, instead, ensure that memory accesses always use the valid copy of the data. The resulting design is open to data replication between the SPMs and the cache hierarchy. The system guarantees that, first, in case of data replication, either the copies are identical or the copy in the SPMs is the valid one and, second, always a valid copy of the data is accessed. For data transfers this is ensured by using coherent DMA transfers and by

guaranteeing that, at the eviction of replicated data, always the invalid copy is discarded and then the valid version is evicted. For data accesses, the compiler identifies memory accesses that may access incoherent copies of data and emits a special form of memory instruction for them. On the hardware side, a set of SPM directories are introduced to track what data is mapped to the SPMs. The execution of these special memory instructions triggers a lookup in the SPM directories, diverting the access to the memory that keeps the valid copy of the data.

4.2.1 Compiler Support

The goal of the compiler in the coherence protocol is to identify memory accesses that may access incoherent copies of data. In order to do this, the compiler support consists on simple modifications to the algorithms of three compiler phases that take part in the process of generating code for tiling software caches: classification of memory references, code transformation and code generation.

4.2.1.1 Classification of Memory References

In this phase the compiler identifies which memory accesses are suitable to be mapped to the SPMs and which others to the GM. It does so by classifying the memory references according to their access patterns and possible aliasing hazards. This last analysis is done using the alias analysis function, which receives two pointers as inputs and gives an outcome with three possible values: the pointers alias, the pointers do not alias or the pointers may alias. The information generated in this phase is added to the intermediate representation of the compiled code and is used in the next phases. The classes of memory references are:

- *Strided accesses* are those that expose a strided access pattern. They are managed by a tiling software cache to access the SPMs.
- *Random accesses* are those that do not expose a strided access pattern and the compiler determines they do not alias with any strided access. They access the GM.
- *Potentially incoherent accesses* are those that do not expose a strided access pattern and the compiler determines they alias or may alias with some strided access. They look up the SPM directories and are diverted to the GM or to some SPM.

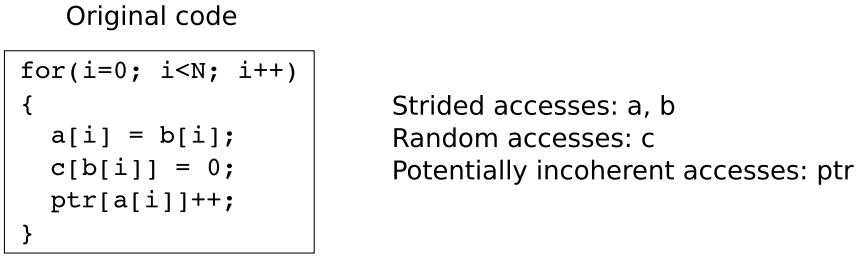

Figure 4.2: Classification of memory references

In the example shown in Figure 4.2 the compiler classifies `a` and `b` as strided accesses because they expose a strided access pattern. Accesses `c` and `ptr` do not follow a strided access pattern so, depending on the outcome of the alias analysis, they are categorized as random or as potentially incoherent accesses. The example assumes the compiler succeeds in ensuring that `c` does not alias with any strided access, so it is classified as a random access, and that the compiler is unable to determine that `ptr` does not alias with any strided access, so it is classified as a potentially incoherent access.

4.2.1.2 Code Transformation

In this phase the compiler performs the code transformations to manage the SPMs using tiling software caches [55, 56, 67, 133]. The code transformations are performed as explained earlier in Section 4.1 and shown in Figure 4.1. The code transformations only operate on the strided accesses, while random accesses and potentially incoherent accesses are not taken into account in this phase. As a consequence, the introduction of the new class of potentially incoherent accesses has no implications in the code transformations.

4.2.1.3 Code Generation

In this phase the compiler takes the transformed code and emits the assembly code for the target architecture, operating on the memory accesses as follows:

- For *regular accesses* the compiler emits conventional memory instructions that directly access the SPMs. This is accomplished by using as source operands the base address of a SPM buffer and an offset.
- For *irregular accesses* the compiler emits conventional memory instructions that directly access the GM. This is accomplished by using as source operands a base address in the GM and an offset.

- For *potentially incoherent accesses* the compiler generates *guarded memory instructions* with an initial GM address. This is accomplished by using as source operands a base address in the GM and an offset. When they are executed, guarded memory instructions look up the SPM directories using the GM address and are diverted to the GM or to some SPM. The implementation of the guarded memory instructions is discussed later in this section.

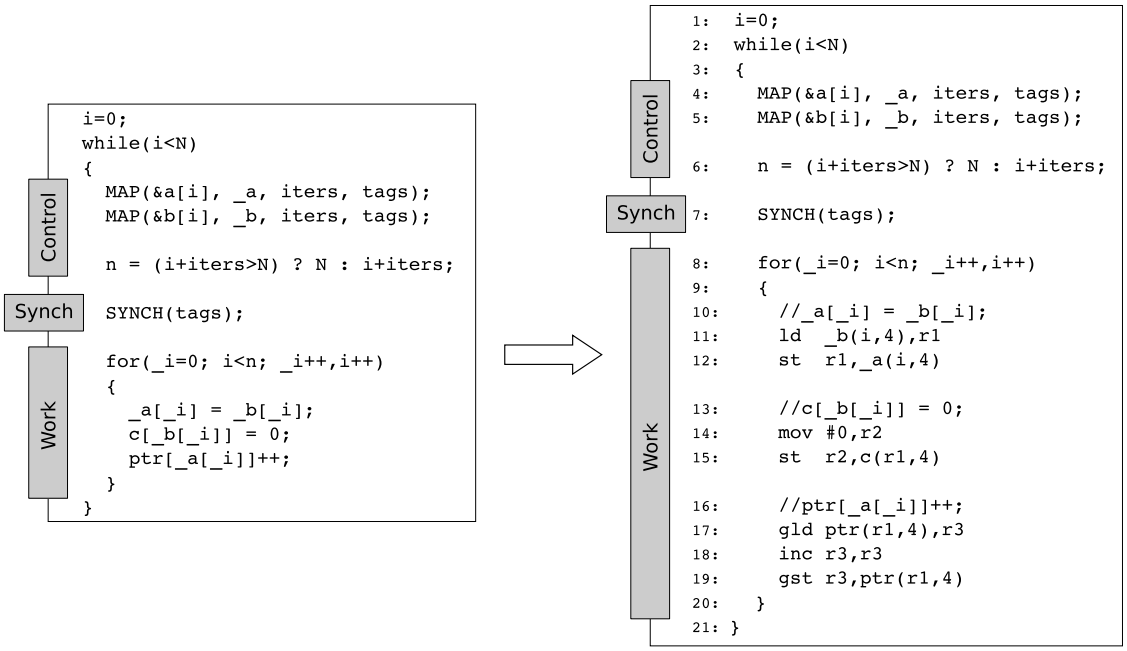

Figure 4.3: Code generation

Figure 4.3 shows the assembly code that the compiler emits for the body of the innermost loop. In the statement that uses strided accesses (line 10), a conventional load (`ld` in line 11) and a conventional store (`st` in line 12) are emitted to, respectively, read a value from `_b` and write it in `_a`. When these instructions are executed, their addresses will guide the memory accesses to the SPMs. Similarly, in the statement that uses a random access to store the zero value in random positions of `c` (line 13), the compiler emits a conventional store (`st` in line 15) with an address that will access the GM at execution time. Finally, to increment the value that is accessed via potentially incoherent accesses (line 16), the compiler emits a guarded load (`gld` in line 17) to read the value and a guarded store (`gst` in line 19) to write the value after incrementing it. When these two guarded memory instructions are executed, the initial GM addresses based on `ptr` will be used to look up the SPM directories and they will be changed to addresses of some SPM if a copy of the data exists there.

The implementation of the guarded memory instructions highly depends on the architecture. The trivial implementation is to duplicate all memory instructions with a guarded form. As this might produce many new opcodes, it may be unacceptable for some ISAs, specially in RISC architectures. One alternative is to take unused bits of the binary representation of memory instructions, as happens in PowerPC [124]. Another option is to provide a fewer range of guarded memory instructions and restrict the compiler to these. In CISC architectures like x86 [76], where most instructions can access memory, instruction prefixes can be used to implement the guard. A generic solution for any ISA is to extend the instruction set by only a single instruction that performs the computation of the address using the SPM directories and leaves the result in a register that is consumed by the memory instruction, conceptually converting the guarded memory access to a coherence-aware address calculation plus a normal memory operation.

4.2.2 Hardware Design

The goal of the hardware support for the coherence protocol is to check if the data accessed by a potentially incoherent access is mapped to some SPM and, in case it is, divert the access to the SPM. With this approach the architecture does not maintain the different copies of the data in a coherent state but, instead, it ensures that the valid copy of the data is always accessed.

The proposed hardware design aims to exploit a key characteristic of the applications: it is extremely rare that, in the same loop, the same data is accessed at the same time using strided and random accesses. The data of a program is kept in data structures, and the internal organization of the data structures is what defines the way the data is accessed, thus defines what kind of memory access are used to do so. In addition, although some data structures (e.g., arrays) can be accessed in different ways, it is unnatural to access the same data using strided and random accesses in the same computational parallel loop. This implies that strided accesses almost never alias with potentially incoherent accesses, although the compiler is unable to ensure it using alias analyses. For this reason, the hardware support for the coherence protocol is designed to not penalize the latency of potentially incoherent accesses that do not access data that is mapped to the SPMs, which is the most common case.

The hardware support for the coherence protocol consists on tracking what data is mapped to the SPMs and what data is known to be not mapped to the SPMs. Figure 4.4 shows the hardware extensions in one core and in one slice of the cache directory of the

cache coherence protocol. Every core tracks what data is mapped to its SPM in its *SPM directory* (shown as SPMDir in the figure), so all the data mapped to all SPMs is tracked in a distributed way. Chunks of data that are not mapped to any SPM and that have been recently accessed by guarded memory instructions are tracked in a hierarchy of filters. Similar to the operation of a directory-based cache coherence protocol, every core tracks some chunks of unmapped data in its *filter* and the *filter directory* tracks the contents of all the filters. Placing the filters and the SPM directories close to the core allows fast access to the information needed to divert potentially incoherent accesses.

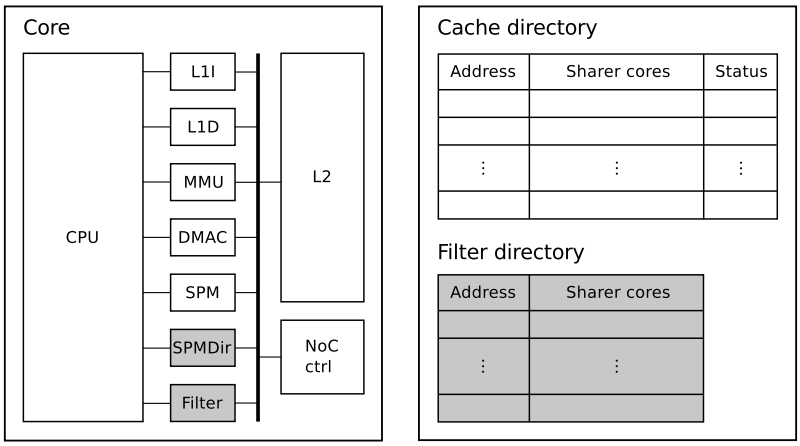

Figure 4.4: Hardware support for the coherence protocol

The next subsections explain the implementation of these hardware structures, how they are operated in the execution of memory operations, and how they track what data is mapped to the SPMs and what data is not mapped.

4.2.2.1 Implementation of Hardware Structures

The hardware structures added for the coherence protocol are implemented as follows:

- The *SPM directory* is a CAM array that tracks the base GM address of the chunks of data mapped to the SPM of the core.
- The *filter* is a CAM array that keeps base GM addresses not mapped to any SPM.
- The *filter directory* is an extension of the cache directory that consists of a CAM array that keeps base GM addresses and a RAM array with a bitvector of sharer cores that tracks which cores have the address in their filters.

Note that the hardware structures track data at a fixed granularity using 64-bit virtual base addresses. This can be done because, in loop parallelism, all threads work with the same SPM buffer size. As explained in Section 4.1, prior to the execution of a loop, the size of the SPM is divided among equally-sized SPM buffers. This buffer size is notified to the hardware, that sets the values of the *Base Mask* and *Offset Mask* internal configuration registers that are used to decompose any address into a base address and an address offset. This allows that, first, base addresses can be used to operate all the hardware structures and, second, the SPM directories do not need a RAM array to store the SPM base addresses for every entry, since the index of the entry is equivalent to the buffer number and, thus, the base address of the SPM buffer.

In other forms of parallelism such as parallel sections, where different threads can simultaneously execute parts of the code that require managing the SPMs at different granularities, range lookups would be required in the proposed hardware structures.

4.2.2.2 Execution of Memory Accesses

The hardware design of the coherence protocol tracks all the data that is mapped to the SPMs in the SPM directories, and the data that has been recently accessed by potentially incoherent accesses that is not mapped to any SPM is tracked in the hierarchy of filters. This information is used in the execution of memory accesses.

Strided accesses are served by the SPM, as shown in Figure 4.5. These kind of memory accesses are the most frequent in HPC applications [107, 152], and they are served in a very power-efficient way because they do not require any CAM lookup in the TLB, the tag array of the caches, the SPM directory or the filter.

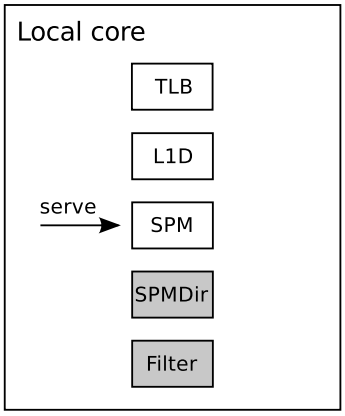

Figure 4.5: Strided access to the SPM

Random accesses are served by the cache hierarchy, as shown in Figure 4.6. These kind of memory accesses operate like in a shared memory multiprocessor without SPMs, accessing the TLB and the L1 cache in parallel. The SPM directory and the filter are not accessed, so no overhead is added.

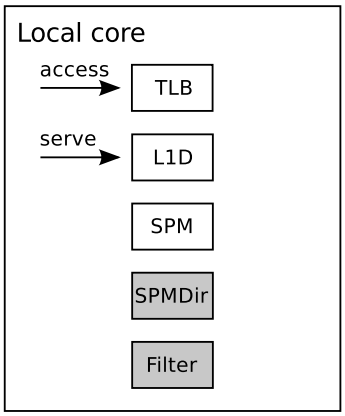

Figure 4.6: Random access to the cache

Potentially incoherent accesses behave as a normal random access to the cache hierarchy, accessing the TLB and the L1 cache. In addition, and in parallel to these two operations, the GM base address is looked up in the filter and in the SPM directory to check if the data being accessed is mapped to some SPM. Different situations can arise depending on the results of the lookups.

Figure 4.7 illustrates the situation where a potentially incoherent access accesses data that is not mapped to any SPM and the address is present in the filter. When this happens the lookup in the SPM directory misses and the lookup in the filter hits ①. The TLB translates the address and the L1 cache is accessed to serve the memory access ②.

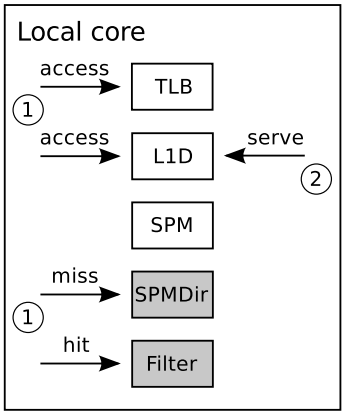

Figure 4.7: Guarded access to data not mapped to the SPMs with filter hit

Figure 4.8 shows the situation where the data accessed by a potentially incoherent access is not mapped to any SPM, but the address is not cached in the filter of the local core. In this case the lookup in the SPM directory misses and the lookup in the filter misses ①, so it has to be checked if the data is mapped to a remote SPM. The cache access is buffered (in the MSHR for loads and in the write buffer for stores) and a request is sent to the filter directory, that looks up the address ②. If the lookup in the filter directory hits it is known that the address is not mapped to any SPM, so the filter directory sends a response to the local core. If the lookup in the filter directory ② misses it has to be checked if the address is mapped to some SPM, so a request is broadcasted to all the remote cores to check if the address is in their SPM directories ③. The lookup misses in all the cores because the data is not mapped to any SPM, and the remote cores respond to the filter directory and this to the local core. When the local core receives the response the buffered cache access is used to serve the memory access ④ and the GM base address is used to update the filter as explained in the next section.

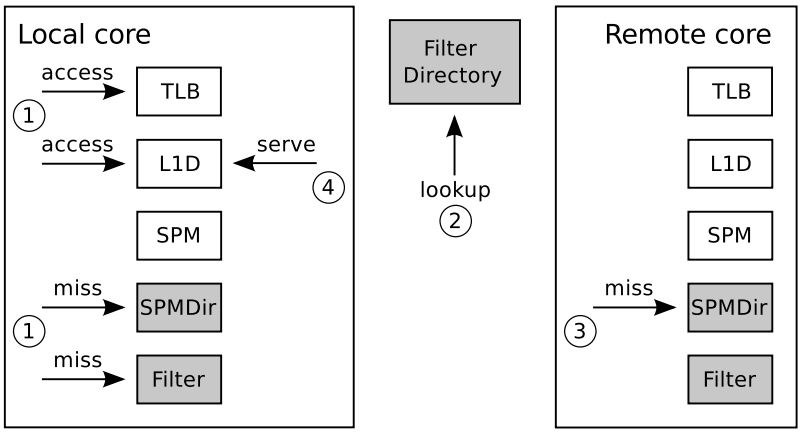

Figure 4.8: Guarded access to data not mapped to the SPMs with filter miss

The situation shown in Figure 4.9 corresponds to the case where the data accessed by a potentially incoherent access is mapped to the SPM of the local core. When this happens the lookup in the filter misses and the lookup in the SPM directory hits ①. The SPM base address corresponding to the SPM directory entry that returned the hit is added to the offset of the address of the guarded instruction, resulting in the SPM address where the data is mapped to. This address is used to access the SPM, which serves the memory access ②. If the potentially incoherent access was a store the data is also written in the L1 cache while, if it was a load, the output of the TLB lookup and the cache access are discarded ③.

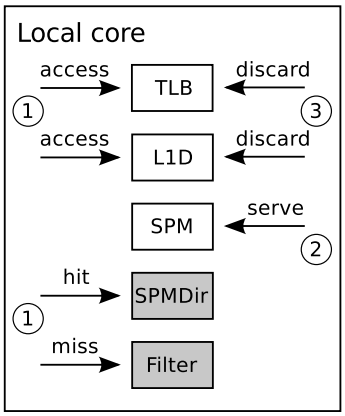

Figure 4.9: Guarded access to data mapped to the local SPM

Figure 4.10 shows the situation where the data accessed by a potentially incoherent access is mapped to the SPM of a remote core. The lookups in the filter and in the SPM directory miss ①, so the cache access is buffered and a request is sent to the filter directory, that also misses ②, and the request is broadcasted to all the remote cores. In this case, one of the remote cores has the data mapped to its SPM, so the lookup in its SPM directory hits ③. The SPM is accessed to serve the remote access ③, calculating the SPM address by adding the offset of the remote access to the base address corresponding to the index of the SPM directory that returned the hit. Then the remote core responds to the filter directory, sending the data if the remote access was a load or an ACK if it was a store, and the filter directory forwards the response to the local core. When the response arrives to the local core the buffered cache access is discarded ④ if it was a guarded load, or the data is written to the L1 cache if it was a guarded store.

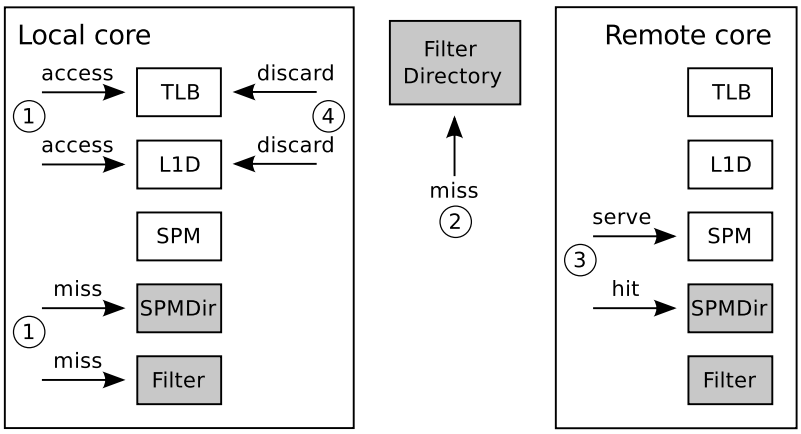

Figure 4.10: Guarded access to data mapped to a remote SPM

4.2.2.3 Tracking SPMs Contents

In order to use the SPM directories, the filters and the filter directory to divert the potentially incoherent accesses to the valid copy of the data, the contents of these hardware structures have to be updated to always track what data is mapped to the SPMs and what data is not mapped.

When a core maps data to its SPM, its SPM directory is updated and the filters are invalidated. Figure 4.11 shows how this process is performed. In order to update the SPM directory ① the source GM address and the destination SPM address of the *dma-get* are used to calculate the GM base address and the SPM base address, and the SPM directory entry associated to the SPM base address is updated with the GM base address. In order to invalidate the filters, first an invalidation message is sent for the GM base address to the filter directory ②, that triggers a lookup of the address ③. If the address is not found no core has the address in its filter, so no more actions take place. If the GM base address is found in the filter directory the entry is invalidated ④ and a filter invalidation message is sent to all the cores in the sharers list ⑤. When the sharer cores receive the invalidation message they look up the address in the filter and invalidate the matching entry ⑥.

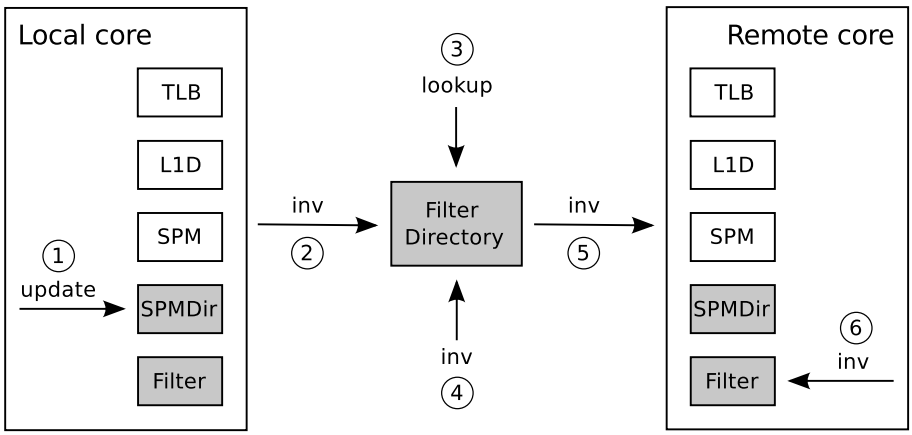

Figure 4.11: SPM directory update and filter invalidation

The filters and the filter directory are updated when the data accessed by a potentially incoherent is not mapped to any SPM, but the address is not present in the filter of the local core. This situation is illustrated in Figure 4.8. The mechanism to update the filters and the filter directory is shown in Figure 4.12, and is very similar to how caches operate on a cache miss. When a lookup of a GM address misses in the filter, it has to check if the GM address is mapped to some SPM. First, a request for the GM address is sent to the filter directory ①, that generates the GM base address and triggers a lookup ②. If

the base address is in the filter directory it means the address is not mapped to any SPM, so the requestor core is added to the sharers list ③ and a NACK is sent as a response ④. When the requestor core receives the NACK it inserts the GM base address in the filter ⑤. If the lookup in the filter directory ② misses, the GM address may be mapped to some SPM. The filter directory broadcasts a request of the address to all the cores ⑥. When the cores receive the request, they calculate the GM base address, look it up in their SPM directory ⑦ and respond to the filter directory either with a NACK if the lookup misses or with an ACK if the lookup hits ⑧. If all the cores respond NACK, the data is not mapped to any SPM, so the filter directory inserts the GM base address ③, sets the local core in the bitvector of sharer cores and sends it a response with NACK ④. When the local core receives the NACK it inserts the GM base address in its filter ⑤. If a remote core responds to the broadcast request with an ACK ⑧ it means it the data is mapped to its SPM, so the address cannot be filtered. The filter directory responds with a ACK to the local core ④ and this does not update its filter.

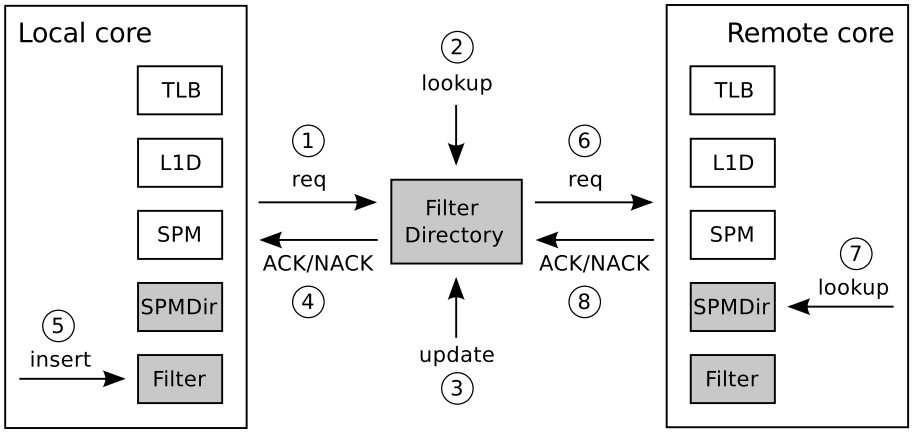

Figure 4.12: Filter update

Note that the filter of every core and the filter directory are associative buffers with a replacement policy so, when they are updated, an older entry can be evicted. When an entry of the filter of a core is evicted, a message is sent to the filter directory to notify of the eviction, and this removes the core from the sharers list. When the filter directory is updated and an older entry is evicted, an invalidation message is sent to all the sharer cores, that invalidate their filter, like in step ② of Figure 4.11.

4.2.2.4 Maintaining Sequential Consistency

Another important problem that affects the ability of the compiler to generate code for the SPMs of the hybrid memory hierarchy is the memory consistency model provided by the architecture. The memory consistency model defines what is the expected behaviour of a sequence of memory operations. Fork-join programming models, as well as the vast majority of shared memory programming models, rely on memory models that imply sequential consistency [1], which ensures that a sequence of memory operations will happen in program order inside the same thread, while a sequence of memory operations from different threads can happen in any order. For this reason, the programmer has to use synchronization mechanisms to avoid data races between the threads, but not between the memory accesses of each thread. In order to allow the compiler to transform the code to manage the SPMs of the hybrid memory hierarchy it is mandatory that the architecture respects the sequential consistency rules.

The coherence protocol for the hybrid memory hierarchy breaks the sequential consistency rules between potentially incoherent accesses and strided accesses of the same thread. The problem is that the virtual address of potentially incoherent accesses is initially a GM address, and it is changed to a different SPM virtual address if the lookup in the SPM directory determines that the data is mapped to the SPM. This causes that, when a thread accesses the same data using strided and potentially incoherent accesses, an out-of-order core can re-order the instructions and the LSQ will not detect an ordering violation has happened because the virtual addresses of the two memory accesses are different. When the potentially incoherent access is sent to the memory hierarchy and its address is changed to point to the SPM it results in the same address of the strided access so, if at least one of the two accesses was a store, the re-ordering breaks the sequential consistency rules. In order to solve this problem, when a potentially incoherent access hits in the SPM directory, the new SPM address is notified to the LSQ to re-check the ordering for the new address, and the pipeline is flushed if a violation is found. If the potentially incoherent access aliases with the contents of a remote SPM it is responsibility of the programmer to synchronize the accesses between the threads, just like it is done for any other type of memory accesses.

4.3 Data Coherence Management

The previous sections describe how memory operations are diverted to one memory or another when replication exists, considering that the valid copy of the data is in the SPMs. This section shows this situation is always ensured. First, the different states and actions that apply to data in the system are described. According to this, it is shown that whenever data is replicated in a SPM and in the cache hierarchy, only two situations can arise: either both versions are identical, or the version in the SPM is always more recent than the version in the cache hierarchy. Then it is shown that whenever replicated data is evicted to main memory, the version in the SPM is always the one transferred, invalidating the cache version. This is always guaranteed unless both versions are identical, in which case the system supports the eviction indistinctly.

4.3.1 Data States and Operations

Figure 4.13 shows the possible actions and states of data in the system. The state diagram is conceptual, it is not implemented in hardware. The *MM* state indicates the data is in main memory and has no replica neither in the cache hierarchy nor in any SPM. The *SPM* state indicates that only one replica exists, and it is located in a SPM. In the *CM* state only one replica in the cache hierarchy exists. In the *SPM-CM* state two replicas exist, one in a SPM and the other in the cache hierarchy.

Actions prefixed with "*SPM-*" correspond to SPM control actions, activated by software. There is a distinction between *SPM-map* and *SPM-unmap* although both actions correspond to the execution of a *dma-get*, which unmaps the previous contents of a SPM buffer and maps new contents instead. *SPM-map* indicates that a *dma-get* transfers the data to the SPM. The *SPM-unmap* indicates that a *dma-get* has been performed that overwrites the data in question, so it is no longer mapped to the SPM. The *SPM-writeback* corresponds to the execution of a *dma-put* that transfers the data from the SPM to the GM. Actions prefixed with "*CM-*" correspond to hardware activated actions in the cache hierarchy. The *CM-access* corresponds to the placement of the cache block that contains the data in the cache hierarchy. The *CM-evict* corresponds to the replacement of the cache block, with its write-back to main memory if needed.

The *MM*→*SPM* transition occurs when the software triggers a *dma-get*, causing a *SPM-map* action. Switching back to the *MM* state occurs when a *SPM-unmap* action happens due to a *dma-get* mapping new data to the SPM buffer. A *SPM-writeback* action

does not imply a switch to the *MM* state, as transferring data to the main memory does not unmap the data from the SPM.

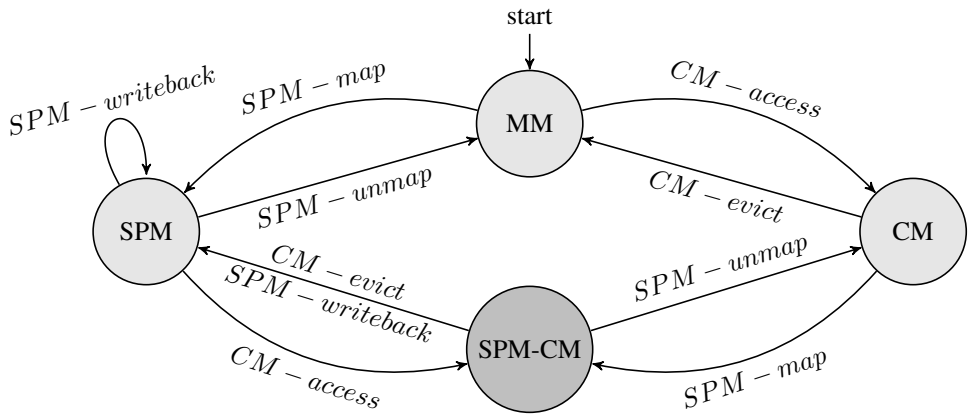

Figure 4.13: State diagram of the possible replication states of the data

Transitions between the *MM* and *CM* states happen according to the execution of random accesses to the GM, which cause *CM-access* and *CM-eviction* actions. Note that unless the data reaches the *SPM-CM* state, no coherence problem can appear due to the use of a SPM. DMA transfers are coherent with the GM, ensuring the system coherence as long as the data switches between the *SPM* and *MM* states. Similarly, the cache coherence protocol ensures the system coherence when the data switches between the *MM* and *CM* states. In both cases, never more than one replica is generated.

The *SPM-CM* state is reachable from both the *SPM* and the *CM* states. In the *SPM* state it is impossible to have random memory accesses to the GM because the compiler never emits them unless it is sure that there is no aliasing, which cannot happen in this state. Potentially incoherent loads never generate a replica in the cache hierarchy since the access goes through the SPM directories, and these divert the access to the SPM. So, in the *SPM* state, only the execution of a potentially incoherent store can cause the transition to the *SPM-CM* state. The execution of a potentially incoherent store writes the data in the SPM and also in the cache hierarchy, so a replica of the data is generated and updated in the cache. Since the same value is written in the two memories, two replicas of the data generated through a *SPM→SPM-CM* transition are always identical.

The transition *CM→SPM-CM* happens due to an *SPM-map* action, and the DMA coherence ensures the two versions are identical. Once in the *SPM-CM* state, potentially incoherent stores updates both versions with the same value, while strided write accesses modify the SPM version and random write accesses are never generated.

In conclusion, only two possibilities exist for having two replicas of data. Each one is represented by one path reaching the *SPM-CM* state from the *MM* state. In both cases, the two versions are either identical or the version in the SPM is the valid one. The next section shows the valid version is always selected at the moment of evicting the data to main memory.

4.3.2 Data Eviction

The state diagram shows that the eviction of data can only occur from the *SPM* and *CM* states. There is no direct transition from the *SPM-CM* state to the *MM* state, which means that eviction of data can only happen when one replica exists in the system. This is a key point to ensure coherence. In case data is in the *SPM-CM* state, its eviction can only occur if first one of the replicas is discarded, which corresponds to a transition to the *SPM* or *CM* states. As seen in the previous section, it is ensured that in the *SPM-CM* state the two replicas are identical or, if not, the version in the SPM is the valid one. Consequently, the eviction discards the cache version unless both versions are identical, in which case either version can be evicted. This behavior is guaranteed by the transitions exiting the *SPM-CM* state. When a *SPM-writeback* action is triggered by a *dma-put* the associated DMA transfer invalidates the version of the data that is in the cache hierarchy. The *CM-evict* transition is caused by an access to some other data in the GM that causes a replacement of the cache block that holds the current data, leaving just one replica in the SPM. So, when transitioning to the *SPM* state, always the replica in the cache is discarded first. Once the *SPM* state is reached, at some point the program will execute a *dma-put* operation to write-back the data to the GM. In the other path, the transition *SPM-CM*$\rightarrow$*CM* caused by a *SPM-unmap* action corresponds to the case where the program explicitly discards the copy in the SPM when new data is mapped to the SPM buffer that holds it. The programming model imposes that this will only happen when both versions are identical, because if the version in the SPM had modifications it would be written-back before being replaced. So, after the *SPM-unmap*, the only replica of the data is in the cache hierarchy and it is valid, and the cache coherence protocol will ensure the transfer of the cache block to the main memory is done coherently.

In conclusion, the system always evicts the valid version of the data. When two replicas exist, first the invalid one is discarded and, then, the DMA and the cache coherence mechanisms correctly manage the eviction of the valid replica.

4.4 Evaluation

This section evaluates the overheads of the proposed coherence protocol and compares the performance, the NoC traffic and the energy consumption of a multicore with 64 cores and the resulting hybrid memory hierarchy against one with a cache hierarchy. Table 4.1 shows the configuration of the hardware structures added for the coherence protocol.

Table 4.1: Configuration of the hardware structures for the coherence protocol

SPM directory	32 entries
Filter	48 entries, fully associative, pseudoLRU
FilterDir	Distributed 4K entries, fully associative, pseudoLRU

4.4.1 Benchmark Characterization

Table 4.2 shows the main characteristics of the benchmarks used in the evaluation. For each benchmark it is shown the input class, the number of kernels, the number of strided and potentially incoherent memory references and the sizes of the data sets accessed by each type of accesses. These benchmarks allow to study many different scenarios regarding the number of memory accesses of each type and the sizes of the data sets they access. CG and IS use few strided references to traverse big input sets and few potentially incoherent references to access a smaller data set, so the ratio of potentially incoherent accesses with respect to the total number of accesses is high. In EP the amount of references of each type is low and the data sets are small and, in addition, its kernels use many scalar temporal values that cause register spilling, so the majority of memory accesses are to the stack. FT and MG traverse big input sets with many strided references and use a few potentially incoherent references to access much smaller parts of the data set, while SP traverses a small input set in a series of 54 different kernels that only use strided accesses.

Table 4.2: Benchmarks and memory access characterization

Benchmark			Strided references		Guarded references	
Name	Input	Kernels	#	Data Size	#	Data Size
CG	ClassB	1	5	109 MB	1	600 KB
EP	ClassA	2	3	1 MB	1	512 KB
FT	ClassA	5	32	269 MB	4	1 MB
IS	ClassA	1	3	67 MB	2	2 MB
MG	ClassA	3	59	454 MB	6	64 B
SP	ClassA	54	497	2 MB	0	0 B

Like in real HPC applications [107, 152], the characteristics of the benchmarks fulfill the motivation for the introduction of SPMs. In all benchmarks the number of strided references is higher than the number of potentially incoherent references, and the data set accessed by strided accesses is much bigger than the one accessed by potentially incoherent accesses. Furthermore, the data sets accessed by strided and potentially incoherent accesses are disjoint, though the compiler is unable to ensure it using alias analyses.

4.4.2 Coherence Protocol Overheads

This section analyses the overheads of the proposed coherence protocol. In this study the coherence protocol is compared with an ideal coherence protocol that diverts potentially incoherent accesses to the correct copy of the data without the need of SPM directories, filters, the filter directory nor any NoC traffic to maintain them. Figure 4.14 shows the overheads in performance, energy consumption and NoC traffic of the hybrid memory hierarchy augmented with the proposed coherence protocol with respect to the hybrid memory hierarchy with ideal coherence.

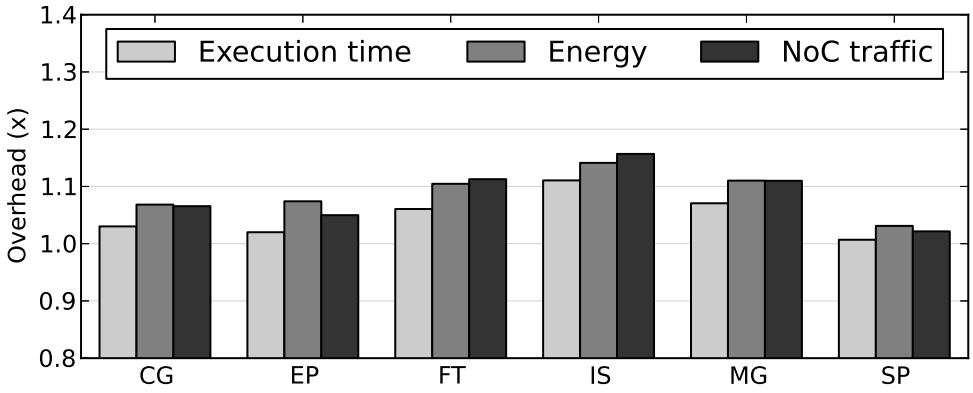

Figure 4.14: Coherence protocol overheads

The coherence protocol adds performance overheads of 1% to 11%. These overheads are caused by the increase in NoC traffic and by the filter misses at the execution of potentially incoherent accesses. The filter hit ratio for each benchmark is shown in Figure 4.15. In SP the performance overhead is negligible because no potentially incoherent accesses are generated and very few DMA transfers are issued, so the filters are not used and the overhead in NoC traffic is only 2%. CG, EP, FT and MG present performance overheads between 2% and 7%. Even though the filter hit ratio for these benchmarks is at least 97%, the NoC traffic added by the coherence protocol, between 4% and 11%, slightly penalizes performance. In IS the performance overhead is 11% because the filter hit ratio is lower,

92%, which penalizes the latency of potentially incoherent accesses and adds higher overheads in NoC traffic, 15%. In all benchmarks the contention in the filter directory is very low due to the low rate of DMA transfers and filter misses. In addition, potentially incoherent accesses never alias with SPM accesses and the filter directory can track the whole data set accessed by potentially incoherent accesses, so filter invalidations and pipeline squashes due to ordering violations never happen.

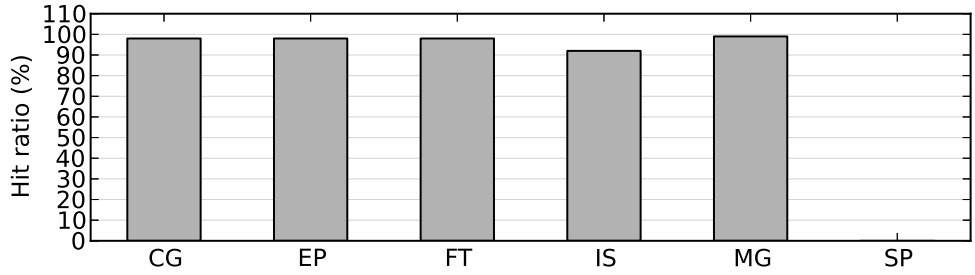

Figure 4.15: Filter hit ratio.

In terms of energy consumption, the overheads range from 3% to 14%. In SP the overhead is only 3% because the filters are gated off and only the SPM directories and the filter directory consume some extra energy at every DMA transfer. In the rest of benchmarks the overheads range between 7% and 14%. Almost half of the overhead is due to static energy consumption, where the SPM directories, the filters and the filter directory add overheads of less than 3%, and the performance overheads add another 2% to 5% in the whole chip. In dynamic energy the SPM directories add up to 3% overhead, while less than 2% is added by the filters and the filter directory. The extra NoC traffic causes overheads of less than 1% in the NoC in all cases except in IS, where it adds 3%. The overhead in area added by the hardware structures for the coherence protocol is 4%.

In conclusion, the coherence protocol introduces very low overheads in performance, energy consumption and NoC traffic, with respective averages of 4%, 9% and 8%.

4.4.3 Comparison with Cache Hierarchies

This section compares the hybrid memory hierarchy against a cache hierarchy in a 64-core multicore architecture. The processor configurations with the hybrid memory hierarchy and with the cache hierarchy have the same characteristics, shown in Table 3.1, but with one difference. For fairness, the L1 D-cache of the architecture with the cache hierarchy is augmented to 64 KB without affecting access latency, matching the 32 KB L1 D-cache plus the 32 KB SPM of the hybrid memory hierarchy.

4.4.3.1 Performance Evaluation

Figure 4.16 shows the performance of the cache hierarchy (Cache) and the hybrid memory hierarchy (Hybrid). Both bars are normalized to the execution time of the cache hierarchy, and show the time spent in each execution phase.

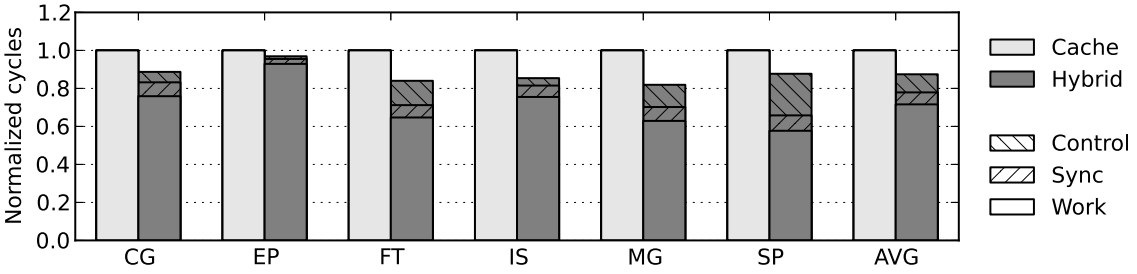

Figure 4.16: Reduction of execution time

The hybrid memory hierarchy reduces the execution time of all benchmarks by 3% to 18% (or speedups of 1.03x to 1.22x). EP is dominated by cache accesses to the stack and a very small data set is mapped to the SPMs, so the performance differences are negligible. In the rest of benchmarks the execution time is reduced by 12% to 18% (or speedups of 1.12x to 1.22x) because the hybrid memory hierarchy is able to serve memory accesses more efficiently than the cache hierarchy. In FT, MG and SP strided accesses dominate, and the hybrid memory hierarchy is able to serve these accesses with the SPMs without performance penalties while, in the cache hierarchy, cache misses happen because the prefetchers are not able to provide all the data for the strided references on time, and also the big amount of data brought and prefetched to the L1 caches causes conflict misses. In benchmarks with potentially incoherent accesses, specially IS and CG but also FT and MG, performance benefits come from exploiting their temporal locality. The hybrid memory hierarchy serves the strided accesses with the SPMs so the data brought to the L1 cache for the potentially incoherent accesses is much less often evicted, while in the cache hierarchy it is quickly evicted to bring and prefetch data for the strided accesses. Moreover, the filters of the coherence protocol present hit ratios of more than 92% in all cases, so the latency of potentially incoherent accesses is not penalized neither. These factors allow the hybrid memory hierarchy to serve data more efficiently than the cache hierarchy, reducing the execution time of the work phase by 25% to 43%. These speedups in the work phase overweight the overheads added in the control and synchronization phases in all cases, resulting on an average reduction in execution time of 13% (or an average speedup of 1.14x).

4.4.3.2 NoC Traffic Evaluation

Figure 4.17 shows the NoC traffic generated by the cache hierarchy (Cache) and by the hybrid memory hierarchy (Hybrid). Both bars are normalized to the NoC traffic of the cache hierarchy and categorize the packets in groups: data cache read, data cache writes (these two groups include data requests, prefetch requests, data and acknowledgment packets), write-back and replacements of cache blocks (WB-Repl, which includes write-back requests, replacements, invalidations, data and acknowledgment packets), DMA transfers (including DMA requests, data and acknowledgment packets) and the coherence traffic introduced by the proposed coherence protocol (CohProt).

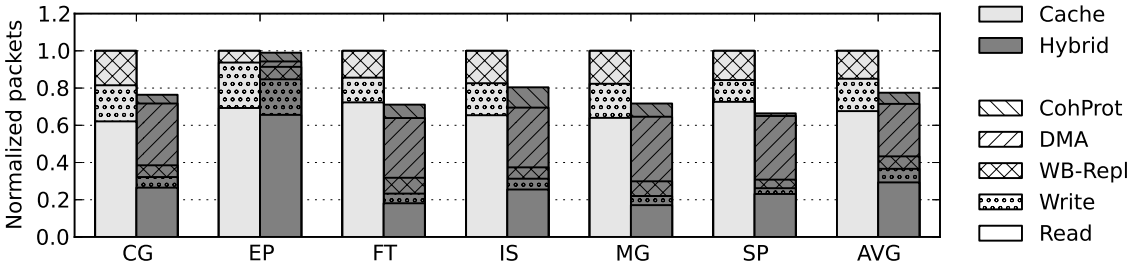

Figure 4.17: Reduction of NoC traffic

The hybrid memory hierarchy reduces the NoC traffic by 20% to 34% in all benchmarks except EP. In these benchmarks most of the data set is mapped to the SPMs and moved using DMA transfers, so many cache accesses, misses and prefetches are avoided, along with their associated NoC traffic. On the one hand, this eliminates between 58% and 75% of the NoC traffic due to data cache reads, between 61% and 74% of the NoC traffic due to data cache writes, and between 41% and 71% of the NoC traffic in the WB-Repl group. On the other hand, the DMA transfers used to move the data add 32% to 37% to the total NoC traffic. Moreover, the runtime system of the tiling software cache causes a higher number of instruction fetches, adding up to 3% of NoC traffic. The proposed coherence protocol also adds NoC traffic, 1% in SP where there are no potentially incoherent accesses, 4% to 7% in CG, FT and MG and up to 10% in IS. EP maps a very small data set to the SPMs and does an intensive utilization of the caches, so the small NoC traffic reductions achieved by mapping data to the SPMs are compensated with the small overhead added by the coherence protocol and the extra code. On average, the hybrid memory hierarchy reduces the NoC traffic by 23%.

4.4.3.3 Energy Consumption Evaluation

Another benefit of the hybrid memory hierarchy is that it consumes less energy than the cache hierarchy, as shown in Figure 4.18. The bars show the energy consumption of the cache hierarchy (Cache) and the hybrid memory hierarchy (Hybrid). Both bars are normalized to the energy consumed by the cache hierarchy, and they also show how the consumed energy is distributed between the CPUs, the caches (including MSHRs, prefetchers and the directories of the cache coherence protocol), the NoC and the memory controllers (NoC + MC), the SPMs and the DMA controllers (SPMs + DMACs), and the structures of the proposed coherence protocol (CohProt).

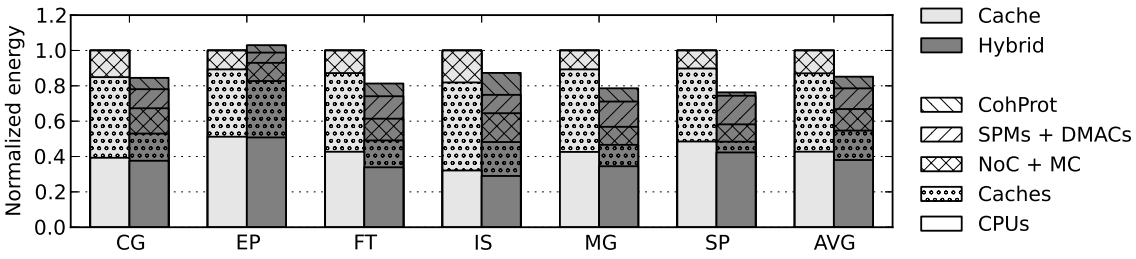

Figure 4.18: Reduction of energy consumption

Results show that the hybrid memory hierarchy achieves energy savings of 13% to 24% in CG, FT, IS, MG and SP. The main reason is the difference in the amount of energy consumed in the caches, which contributes with more than 41% of the total energy consumed in the cache hierarchy and is reduced by a factor of 2.3x to 6.7x in the hybrid memory hierarchy. This happens because most of the memory accesses are served by the SPMs, that consume between 10% and 16% of the total energy to do so. The hardware structures of the proposed coherence protocol consume between 6% and 12% of the total energy except in SP, where it represents only 1% because no potentially incoherent accesses are used. The energy consumed in the CPUs is also reduced, between 5% and 21% depending on how many instruction re-executions due to cache misses are avoided, while the reduction in NoC traffic also reduces by 18% to 32% the energy consumed in this component. In EP the SPMs of the hybrid memory hierarchy are underutilized and the static energy of the added structures causes an overhead of 3%. On average, the energy saved by the hybrid memory hierarchy in all benchmarks is 15%.

4.5 Summary and Concluding Remarks

This chapter proposes a coherence protocol that allows the compiler to automatically generate code to manage the SPMs of the hybrid memory hierarchy from fork-join programming models. SPMs are more power-efficient than caches and they do not generate coherence traffic but they impose programmability difficulties. Giving the compiler the responsibility of generating code to manage the SPMs is a promising solution to overcome this programmability issues.

Tiling software caches are a well-known technique to efficiently manage SPMs. In this approach the data accessed by strided accesses is mapped to the SPMs, taking advantage of the predictability of these memory accesses to manage the data in software with low overheads. However, unpredictable random accesses are very difficult to manage by software, so they are served by the cache hierarchy. Although compilers can generate code to manage the SPMs with tiling software caches in trivial cases, they are unable to do so in the presence of pointers with unknown memory aliasing hazards.

In order to avoid this problem, this chapter proposes a hardware/software co-designed coherence protocol where the compiler identifies potentially incoherent memory accesses that may access incoherent copies of data, and generates a special form of memory instruction for them. On the hardware side, the coherence protocol consists of a set of directories and filters that track what data is mapped to the SPMs and what data is not mapped to any SPM. These structures are used to determine if the potentially incoherent accesses have to be served by the SPMs or by the cache hierarchy, ensuring the valid copy of the data is always accessed.

With the proposed coherence protocol the compiler can always generate code to manage the SPMs of the hybrid memory hierarchy from fork-join programming models, so the architecture can be exposed to the programmer as a shared memory multiprocessor and the goodnesses of the hybrid memory hierarchy can be capitalized without any programming burden. The ability of the hybrid memory hierarchy to serve memory accesses with the SPMs and to efficiently move the data with DMA transfers provides important benefits when compared to a cache hierarchy. In a multicore with 64 cores, an average speedup of 1.14x is achieved due to the efficiency on serving memory accesses without performance penalties. In addition, using DMA transfers leads to an average reduction in NoC traffic of 23%, and serving most of the memory accesses with the SPMs instead of the caches is the biggest factor to reduce the energy consumption by an average 17%.

Chapter 5

SPM Management in Task Models

Task programming models are a very appealing approach to program future heterogeneous architectures. In these programming models the programmer exposes the available parallelism of an application by splitting the code in tasks and by specifying the data and control dependences between them. With this information the runtime system manages the parallel execution of the workload, scheduling tasks to cores and taking care of synchronization between tasks.

Another benefit of task programming models is that decoupling the application from the hardware allows to apply many optimizations at the runtime system level in a generic and application-agnostic way [146, 34]. For instance, the task scheduler can not only ensure load balancing, but also aim for a locality-aware schedule [28]. Moreover, in the task data-flow paradigm the runtime system knows what data is going to be accessed by the tasks before they are executed, which enables multiple optimizations like data prefetching [119, 61] or efficient data communication between tasks [100, 101].

This chapter proposes to take advantage of the performance, scalability and power consumption benefits of hybrid memory hierarchies in task data-flow programming models, using the task annotations to manage the SPMs in the runtime system without affecting programmability. To do so, the runtime system exploits the characteristics of task data-flow programming models to map the data specified in the task dependences to the SPMs, so memory accesses to this data are served in a power-efficient way and without generating coherence traffic, while the rest of memory accesses are served by the cache hierarchy. The runtime system can also perform locality-aware scheduling, minimizing the data movements in the memory hierarchy by assigning tasks to the cores that already have the data of the task dependences mapped to their SPMs. In addition, the runtime system incorporates mechanisms to mitigate the communication costs of the DMA transfers for the SPMs, overlapping them with useful computation such as the task scheduling

phases or the execution of previous tasks. Since the size of the SPMs has a direct impact on the granularity of the tasks, the study also contemplates pairing the proposed SPM management strategies with different SPM sizes and task runtime systems with hardware support.

5.1 Suitability

Task data-flow programming models are specially well suited for SPMs. The specification of the input and output dependences for the tasks provide the runtime system with the information of what data is going to be accessed, which allows to map the tasks input and output dependences to the SPMs. As a consequence, memory accesses to inputs and outputs will always access the SPMs during the execution of tasks. Figure 5.1 shows the distribution of memory accesses for a set of representative benchmarks. The figure shows, for each benchmark, the percentage of loads and stores that access data specified in task dependences (Dep loads and stores) and the percentage of loads and stores that access other memory locations (Other loads and stores). The benchmarks present a wide range of percentages of memory accesses to task dependences, from 0% in `raytrace` and `fluidanimate` to 76% in `md5`. On average, 40% of the memory accesses are to task inputs and outputs, so a significant amount of memory accesses can be efficiently served by the SPMs. In particular, compared to cache accesses, memory accesses to the SPMs do not suffer performance penalties in the form of cache misses, they consume less power because they do not trigger lookups in the tags of the caches nor in the TLBs, and they do not generate coherence traffic.

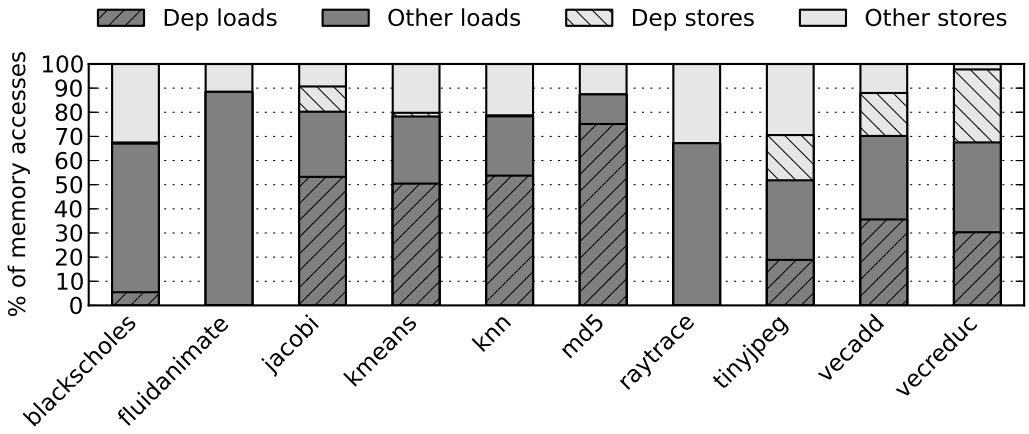

Figure 5.1: Percentage of memory accesses to tasks dependences

The memory model of task data-flow programming models is another very important factor for the suitability of SPMs. The memory model of task programming models guarantees that, during the execution of a task, its inputs will not be modified by another task and its outputs will not be accessed by another task. This property effectively eliminates the data races to the input and output dependences, so there is no need to maintain coherence for this data during the execution of a task. This allows that the data specified in the task dependences can be safely mapped to the SPMs during the execution of a task without requiring any costly synchronization mechanism in these non-coherent memories.

Additionally, the execution model of task programming models offers the possibility to hide the communication costs of DMA transfers. First, the runtime system can perform scheduling decisions to exploit data locality, aiming to reduce data motion by assigning tasks to a core that already has the dependences mapped to its SPM. Second, when locality cannot be exploited, the runtime system can trigger the DMA transfers for the task dependences before the task is executed, so the communication is overlapped with other execution phases such as the task scheduling phase or the execution of the previous task.

5.1.1 Suitability of Other Programming Models

Besides purely task models, other programming models designed for heterogeneous architectures are good candidates to transparently manage the SPMs of hybrid memory hierarchies. Offload models like OpenHMPP [49] or OpenACC [113] also use source code annotations and clauses that allow to specify what data has to be copied from the host processor memory to the accelerator memory, they expose similar memory models in terms of the privateness of the data during the execution of the kernels and they also use a runtime system to orchestrate the data transfers and kernel executions. Thanks to these properties, the code annotations can also be exploited in these models to map data to the SPMs of a hybrid memory hierarchy. Moreover, opportunities to hide the cost of the data transfers are also found in offload models, like in OpenAcc, that supports clauses to allow asynchronous data transfers and to specify at which point the execution should wait for all the asynchronous data transfers to be completed.

Although the proposal of this chapter focuses on how to automatically manage the SPMs of a hybrid memory hierarchy from the runtime system of task data-flow programming models, the proposed ideas can be easily adapted to other parallel programming models with similar characteristics, so the contributions are applicable to a wide range of programming models and applications.

5.2 SPM Management in Task Runtime Systems

The goal of the runtime system is to transparently manage the SPMs of the hybrid memory hierarchy. This section describes what data structures are added in the runtime system and how they are operated to map task dependences to the SPMs. In addition, it is explained how the runtime can perform optimizations such as locality-aware scheduling and overlapping of DMA transfers with computation.

5.2.1 Mapping Data Dependences to the SPMs

The typical behaviour of a thread in a task runtime system for shared memory multiprocessors is an iterative process that consists on requesting a task to the scheduler, executing the task and waking up its dependent tasks. This behaviour is shown in Figure 5.2 in the timeline labeled as `Cache`.

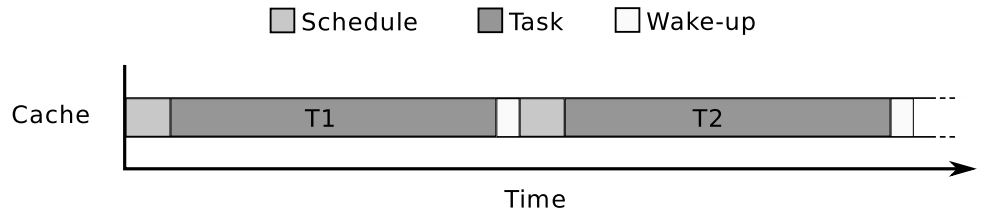

Figure 5.2: Timeline of a task application with the Cache behaviour

In the scheduling phase the thread running on a core requests a new task to the scheduler. The scheduler selects a task from the ready queue based on a certain policy[1], removes the task from the ready queue and passes its associated task descriptor to the requesting thread. The task descriptor includes information about the task such as a pointer to the function that encapsulates the code or the addresses of the dependences, that are passed to the function as parameters when the task is executed. When the task finishes, the scheduler wakes up its dependent tasks. The scheduler locates in the TDG the node that represents the task that has just been executed and, for every out-going edge representing an output dependence, marks as ready the in-going edge of the neighbour node, which represents an input dependence of a dependent task. When an input dependence of a task is marked as ready the scheduler checks if all the other input dependences of the task are

[1]The default policy is First-In First-Out (FIFO), but Section 5.2.3 presents other policies aware of data locality

also ready, so it can be woken up. In the `Cache` behaviour the scheduler wakes up ready tasks by inserting them in the ready queue.

For the hybrid memory hierarchy, four phases are added to this execution model to map the task dependences to the SPM of the core. The phases are *map inputs* and *synchronize inputs* before the execution of a task and *map outputs* and *synchronize outputs* after the execution of a task. In addition, several data structures are added in the runtime system to operate in these phases. Figure 5.3 shows the extensions in the runtime system, where added data structures are shaded in gray. Apart from the described ready queue and TDG, the scheduler requires a *Dependents List* to perform data locality-aware schedulings. Section 5.2.3 further describes this extension. Next, each core abstraction in the runtime system has an associated thread that is pinned to a physical hardware thread, and a task descriptor of the currently executing task. A new per-core data structure, the *SPM directory*, is added to manage the mapping of inputs and output to the SPM. The SPM directory keeps, for every dependence mapped to the SPM of the core, the base address of the copy of the data in the SPM. Finally, a *Next Task Descriptor* is required to perform double buffering of DMA transfers with task execution, as described in Section 5.2.2.

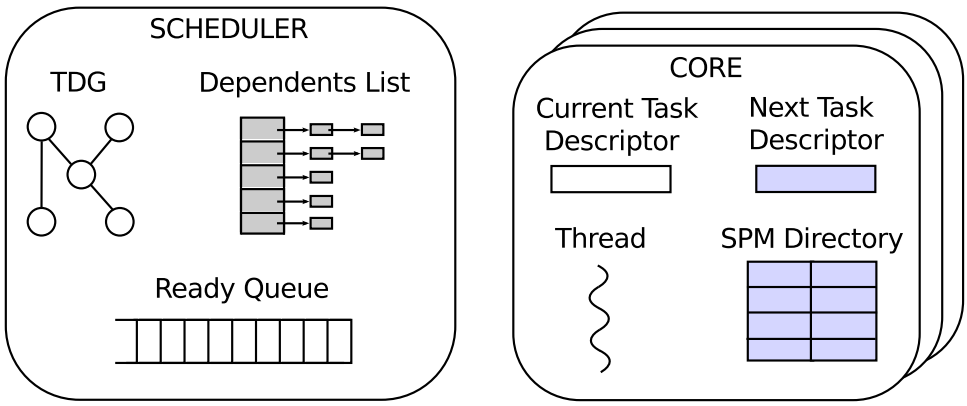

Figure 5.3: Runtime system extensions to support hybrid memory hierarchies

Figure 5.4 shows the behaviour of a task workload on the hybrid memory hierarchy when DMA transfers are not overlapped with any computation, labeled as `SPM-NoOv`. In the map inputs phase, once a task has been scheduled on a core, the task dependences are mapped to the SPM of the core. First, for each entry in the SPM directory of the core, it is checked if the mapping matches any dependence of the task. If there is no match the SPM directory entry is erased and the space in the SPM is freed while, if a match is found, the task dependence is marked as already mapped. Then, for every task dependence that is not already mapped to the SPM, the necessary space is allocated for it in the SPM, the

new mapping is recorded in the SPM directory and a DMA transfer is issued to copy the data to the SPM. Note that the data for an output dependence is also brought to the SPM because, if only some parts of the chunk of data are modified, the write-back at the end of the task execution will update the copy of the data in GM with wrong values. Once all the dependences are mapped to the SPMs the pointers that are passed to the task for the inputs and outputs are changed, substituting the original pointers in the task descriptor for the pointers to the data in the SPM.

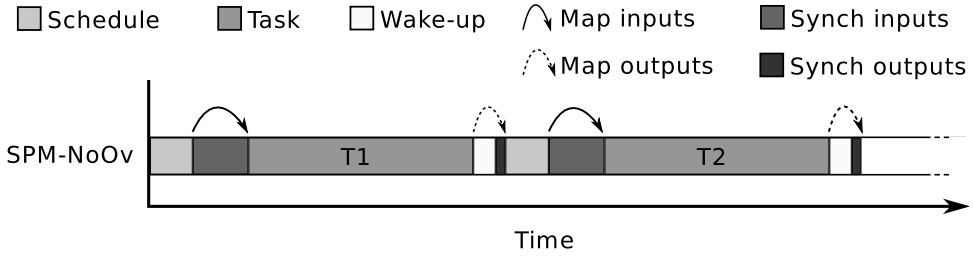

Figure 5.4: Timeline of a task application with the SPM-NoOv behaviour

In the synchronize inputs phase, just before the task starts executing, the thread waits for the DMA transfers of the task dependences to finish. When these DMA transfers have finished the thread jumps to the code of the task to start its execution. During the execution the new pointers to the SPM mappings ensure that memory operations to the inputs and outputs access the address space of the SPM, so this memory serves the accesses.

At the end of the task execution the map outputs phase takes place. In this phase the thread consults the SPM directory and, for each output dependence of the task, a DMA transfer is triggered to write back the results to the GM. Note that, even in the case that the output dependence is going to be reused as input by the following task executed on the core, the DMA transfer to write back the data to the GM is still done because other tasks that also reuse the output dependence as input may be executed on other cores.

While the data of the output dependences is written back, the scheduler wakes up the tasks that depend on these dependences using the TDG. The main difference with the already explained behavior is that new ready tasks are kept apart from the ready queue until the write-back DMA transfers finish.

Finally, in the synchronize outputs phase, the thread synchronizes with the write-back DMA transfers for the output dependences of the task that has just been executed. When the DMA transfers have finished, the scheduler finally inserts the tasks that were woken up in the previous phase in the ready queue. Then the thread repeats the whole process to execute the next task.

5.2.2 Overlapping DMA Transfers with Computation

The DMA transfers triggered by the runtime system to manage the SPMs may impose high overheads in the synchronization phases if they are not overlapped with any computation phase. This section explains two mechanisms to reduce the impact of the communication cost of the data transfers for the SPMs. The solutions consist on overlapping the DMA transfers with the scheduling phase, denoted `SPM-RT`, and double buffering with the execution of the previous task, denoted `SPM-DB`.

For both approaches the runtime system needs to assign two tasks per core instead of one. For this purpose a new element *Next Task Descriptor* is added in the core abstraction of the runtime system, which keeps the task that is going to be executed by the core after the *Current Task Descriptor*.

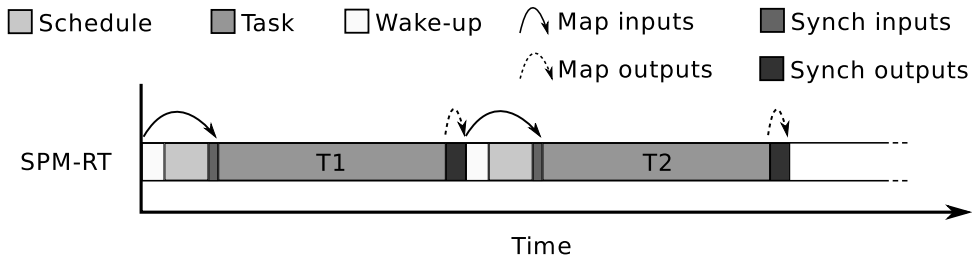

Figure 5.5: Timeline of a task application with the SPM-RT behaviour

The first approach consists on overlapping the DMA transfers with the task wakeup and scheduling phases. This behaviour is shown in Figure 5.5 in the timeline labeled as `SPM-RT`, which shows how the phases for the execution of two tasks T1 (the current task) and T2 (the next task) are interleaved. Task T1 starts by copying its dependences to the SPM of the core in the map inputs phase. While the data for T1 is being transferred using DMA transfers the wakeup phase of the previous task T0 takes place, which marks as ready the tasks that depend on T0. Then the thread requests a task to be executed after T1 to the scheduler, which assigns task T2 to the core. This task T2 is kept in the next task descriptor field of the core abstraction in the runtime system. Note that, since the map inputs phase of T1 has already happened, the mappings for T1 are already present in the SPM directory of the core when the next task T2 is requested, so the locality-aware scheduler takes into account the data mapped by T1 although it has not been yet executed. Once T2 has been scheduled as the next task on the core, the synchronization with the inputs of T1 takes place and the task is executed normally. Just after the task T1 finishes its execution, its outputs are written back to GM in the map outputs phase and the thread

waits for the write-back to finish in the synchronize outputs phase. At this point the runtime system triggers the DMA transfers of the map inputs phase of the next task T2, so they are overlapped with the wake-up phase of T1 and the scheduling phase of the task that is going to be executed after T2.

The second approach is a double buffering technique that overlaps the DMA transfers for a task with the execution of the previous task. The timeline labeled as `SPM-DB` in Figure 5.6 shows this behaviour for the execution of two tasks, the current task T1 and the next task T2. The timeline shows that the succession of phases is the same as in the `SPM-NoOV` behaviour, with the only difference that the map input phases are not for the task that is about to be executed but for the following one. The timeline starts with the map inputs phase of the next task T2. While the dependences for T2 are being transferred to the SPM, the thread waits for the inputs of the current task T1 (its map inputs phase is not shown because it happened before the execution of the previous task), executes the task, copies the outputs, calls the scheduler to wake up its dependent tasks and to schedule a new task T3 and synchronizes with the output DMA transfers. Then, this process is repeated for T2, which gets executed while the inputs of T3 are transferred, and for the subsequent tasks.

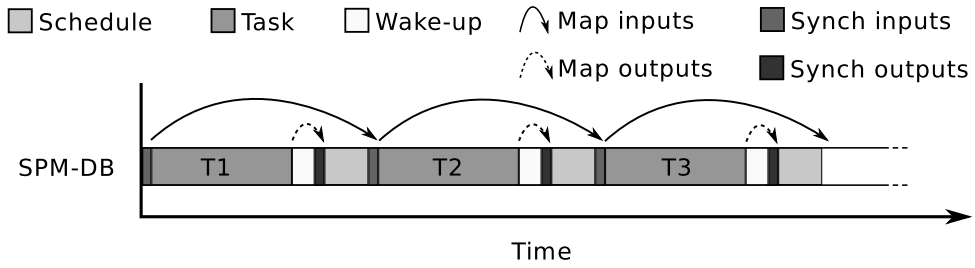

Figure 5.6: Timeline of a task application with the SPM-DB behaviour

5.2.3 Locality-Aware Scheduling

The task scheduler is a fundamental part of a task runtime system. As explained in the previous section, when a thread wants to execute a task it first requests a new task to the scheduler, which selects a task from the ready queue based on a given policy. The locality-aware scheduler selects tasks for execution aiming to minimize the amount of data that has to be moved in the memory hierarchy. This scheduler can be used to minimize the number of DMA transfers in the hybrid memory hierarchy and also to improve data locality in traditional cache hierarchies.

The locality-aware task scheduler uses an additional data structure, the *dependents list*, as shown in Figure 5.3. The dependents list tracks, for a given dependence, what are the ready tasks that depend on it. This data structure is used by the locality-aware scheduler to quickly identify tasks in the ready queue that depend on a given dependence, avoiding a traversal of the ready queue.

The dependents list is updated when a task is inserted or erased from the ready queue. When the scheduler inserts a task in the ready queue it checks, for all the dependences of the task, if they are present in the dependents list. If the dependence is found the task is inserted in the list associated to that entry, otherwise a new entry for the dependence is created along with an empty list, in which the task is then inserted. When the scheduler assigns a task to a core it removes the task from the ready queue and traverses, for each dependence of the task, its associated dependents list to remove the task from the list.

When a core requests a new task to the scheduler this selects from the ready queue the task that already has more data mapped to the SPM of the core. In order to do this the SPM directory of the core is traversed and, for every dependence already mapped to the SPM, its dependents list is accessed to obtain a list of ready tasks that reuse the dependence. The scheduler selects from this list the ready task that has more data mapped to the SPM to be executed on the core. If the data present in the SPM is not a dependence of any ready task the scheduler selects the task at the head of the ready queue.

5.2.4 Discussion

The proposed mechanisms allow the runtime system to map task dependences to the SPMs of the hybrid memory hierarchy. As a first approach, the proposal assumes that the size of the task dependences is always smaller than the available space in the SPMs for them. Under this assumption it is the programmer who has to ensure that the data for the task dependences fits in the SPMs so, when the application is divided in tasks, this restriction has to be taken into account. To allow programmers to taskify their codes without this restriction several solutions can be applied at the runtime system level. An straightforward solution is to discard mapping the data for the dependences that do not fit in the SPM, so they are served by the cache hierarchy. Since this solution may end up underutilizing the SPMs, some other approaches could be studied, such as performing automatic task coarsening in the runtime system to fuse or split tasks according to the available space in the SPM, or including a lightweight pagination mechanism for the SPMs so parts of the input and output dependences are mapped and unmapped on demand.

The two proposed overlapping techniques have different trade-offs. On the one hand, the execution of a task is usually longer than only the wake-up and scheduling phases, so the double buffering with the previous task has more time to overlap the DMA transfers. On the other hand, doing double buffer with the previous task imposes that the available space in the SPM has to be shared by two tasks. Due to this restriction, and depending on how the application is split in tasks, more tasks may be needed to perform the same amount of work, which can incur in higher runtime system overheads [57, 89, 127].

Finally, task programming models themselves have some limitations. Data structures with pointers and indirections are hard to handle by task programming models, specially in those where dependences are statically declared using pragmas. In addition, in shared memory multicores it is not strictly necessary to specify all the data produced and consumed by the tasks as dependences, so programmers some times only specify the minimum amount of dependences that ensure the execution is correct, or introduce additional variables to synchronize tasks manually. This kind of bad programming practices can also cause an underutilization of the SPMs in some cases.

5.3 Evaluation

This section evaluates the proposed runtime system techniques to manage the SPMs, comparing a multicore with 32 cores and a hybrid memory hierarchy against one with a cache hierarchy. For fairness, the L1 D-cache of the architecture with the cache hierarchy is augmented to 64 KB without affecting its access latency, matching the 32 KB L1 D-cache plus the 32 KB SPM of the hybrid memory hierarchy.

5.3.1 Performance Evaluation

The normalized execution time of the hybrid memory hierarchy with respect to the cache hierarchy is shown in Figure 5.7. The execution time of the three proposed data transfer strategies (SPM-NoOv, SPM-RT and SPM-DB) are evaluated and, moreover, an ideal configuration (SPM-Ideal) where DMA transfers occur instantaneously is also shown. All results are normalized against the cache configuration, so values below 1 represent reduction in execution time. This figure further distinguishes how the execution time is distributed between phases: execution of tasks (`Task`), synchronization with DMA transfers (`Sync`), which includes the synchronize inputs and outputs phases, and runtime (`Runtime`), that includes the wake-up, scheduling and map inputs and outputs phases.

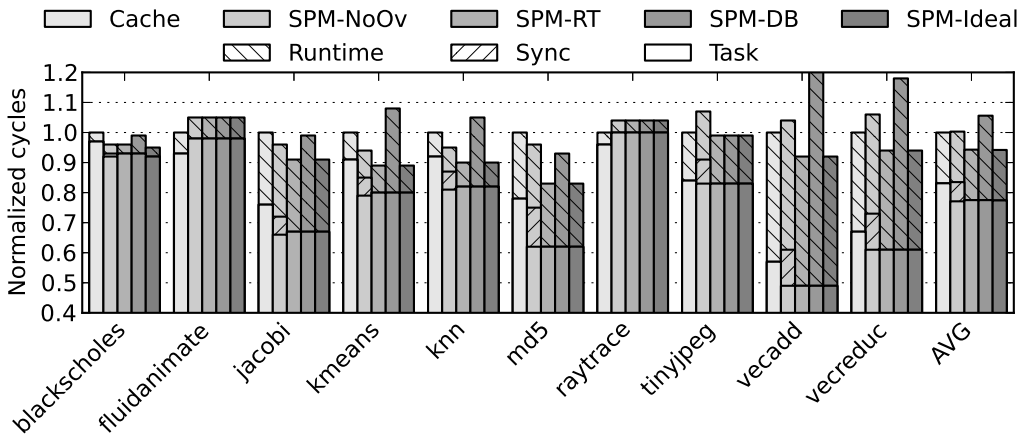

Figure 5.7: Reduction of execution time

The task execution phases are accelerated in all benchmarks except `fluidanimate`, `raytrace` and `tinyjpeg`, achieving an speedup of up to 22% (`md5`). This happens because task dependences are served by the SPMs in the hybrid memory hierarchy, so performance penalties due to cache misses are minimized. When the cache hierarchy presents close to 100% hit ratio in the L1 D-cache (`tinyjpeg`) no performance improvements are observed in the execution of tasks. In benchmarks that do not map data to the SPMs (`fluidanimate` and `raytrace`) the performance in the task execution phases decreases because of the augmented L1 D-cache in the cache hierarchy baseline. These performance improvements in the task execution phases allow the hybrid memory hierarchy to achieve up to 5% speedup if DMA transfers are not overlapped. In this SPM-NoOv approach, when big amounts of data are mapped to the SPMs, the synchronization time adds overheads of up to 11% (`tinyjpeg`), limiting the performance of the hybrid memory hierarchy. On average, the cache hierarchy and the hybrid memory hierarchy offer the same perfomance if DMA transfers are not overlapped. The SPM-RT strategy achieves an average speedup of 6%, reaching up to 16% in `md5`. In the SMP-RT approach the time spent in the synchronization phases becomes negligible in all cases, so the performance is very close to the one of the ideal configuration. For SPM-DB the synchronization time also becomes negligible but the number of executed tasks increases together with the runtime system overhead in some cases. As a consequence, the time spent in runtime phases increases significantly in some benchmarks (`jacobi`, `kmeans`, `vecadd` and `vecreduc`) and neglects the performance benefits of using the SPMs. In other benchmarks (`md5`) the double buffering does not cause a big increase of the runtime system overhead, resulting in a speedup of 6%.

5.3.2 NoC Traffic Evaluation

Another important benefit of hybrid memory hierarchies is the reduction of NoC traffic. Figure 5.8 presents the reduction in NoC traffic with respect to the cache hierarchy. Each bar shows the percentage of traffic originated by different actions: cache reads and writes (which include packets for data requests, prefetch requests, data and acknowledgements), write-back and replacement of cache lines (Wb-Repl, which includes packets for write-back requests, replacements, invalidations, data and acknowledgements), and DMA transfers (which include packets for DMA requests, data and acknowledgements).

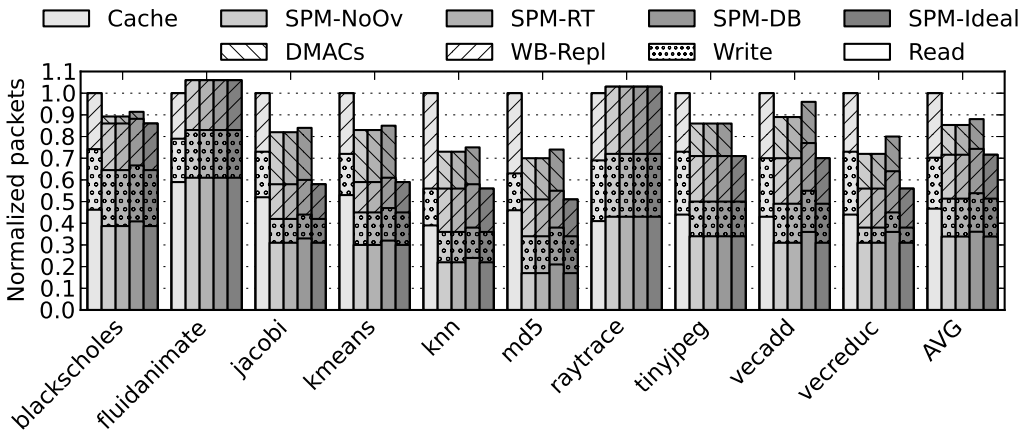

Figure 5.8: Reduction of NoC traffic

The hybrid memory hierarchy reduces the Noc traffic related to cache reads, writes and WB-Repl in all configurations. This reduction is proportional to the percentage of mapped accesses to the SPMs, reaching up to 62% reduction of read traffic in `md5` as most of the loads access task dependences. Similarly, the NoC traffic originated by cache writes is reduced if output dependences are mapped to the SPMs, achieving savings of up to 39% in `jacobi` in this category, although the average reduction is 18% as the portion of writes mapped to the SPMs is smaller than in the case of loads. The reduced activity in the caches also reduces cache misses, replacements and invalidations, so the traffic in the WB-Repl group is reduced between 17% (`blackschoes`) and 59% (`md5`). In the hybrid memory hierarchy all this NoC traffic is saved thanks to the introduction of SPMs, that needs DMA transfers to move the data. The NoC traffic generated by DMA transfers to move the task dependences contributes with less than 30% of the original traffic in all cases, and never overweights the traffic saved in the other categories. Consequently, an average reduction in NoC traffic of 15% is obtained with the hybrid memory hierarchy.

5.3.3 Energy Consumption Evaluation

Figure 5.9 shows the reduction in energy consumption of the hybrid memory hierarchy with respect to the cache hierarchy. Two bars are shown for each benchmark, one for the cache hierarchy (Cache) and one for the hybrid memory hierarchy (SPM-NoOv). All results are normalized to the energy consumption of the cache hierarchy, so values below 1 represent a reduction in energy consumption. All the data mapping techniques (SPM-NoOv, SPM-RT and SPM-DB) present the same trends, so only one bar is shown for the hybrid memory hierarchy. The figure also shows how the energy consumption is distributed among the different components of the architectures: cores (CPUs), caches, prefetchers, MSHRs and cache directories (Caches), the network-on-chip and the memory controller (NoC + MC), and the SPMs and the DMA controllers (SPMs + DMACs).

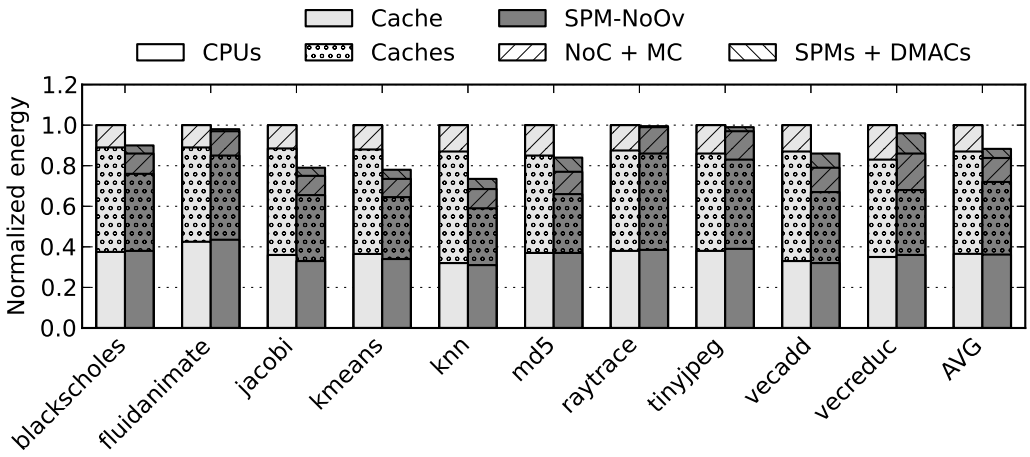

Figure 5.9: Reduction of energy consumption

Results show that the energy consumed by the CPUs is nearly the same in both architectures. It can be observed a reduction of the energy consumed in the caches, with savings of up to 50% in benchmarks that map a significant portion of accesses to the SPMs (`jacobi`, `kmeans`, `knn` and `md5`). The big energy savings in these components happen because, in the hybrid memory hierarchy, many memory accesses are served by the SPMs instead of the caches. SPMs are able to serve these memory accesses in a much more energy-efficient way, contributing with less than 10% of the total energy consumed for all benchmarks. In all cases, the overall energy consumption in the components of the memory hierarchy (Caches and SPM+DMACs) is lower in the hybrid memory hierarchy than in the cache hierarchy, resulting in an average reduction in energy consumption of 12%.

The speedup in Energy Delay Product (EDP) of the hybrid memory hierarchy with respect to the cache hierarchy is shown in Figure 5.10. The hybrid memory hierarchy achieves speedups in EDP in almost all configurations, with average improvements of 14%, 29% and 0% for SPM-NoOv, SPM-RT and SPM-DB, respectively. These improvements are particularly significant in the benchmarks that map more data to the SPMs, achieving up to 59% improvement in EDP in `knn` with the SPM-RT configuration. Some benchmarks present slowdowns in EDP caused by the performance overheads in the runtime system in the SPM-DB configuration and by the synchronization time spent in SPM-NoOV configurations.

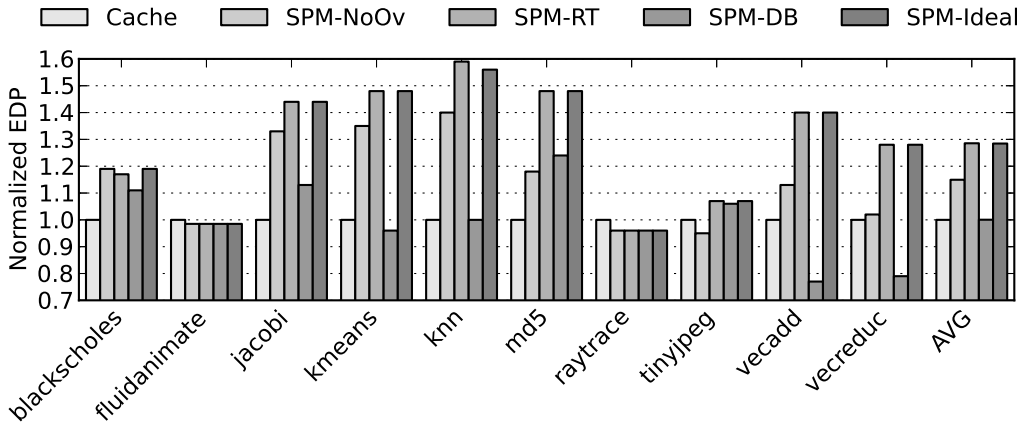

Figure 5.10: Speedup in EDP

5.3.4 Mitigating the Effects of Fine-Grained Tasks

It has been shown that the overhead in the runtime system can degrade performance when fine-grained tasks are required. This is an important factor for the hybrid memory hierarchy, as the size of the SPMs determines the granularity of the tasks.

One way to alleviate the runtime system overheads is to increase the size of the SPMs. Figure 5.11 shows the average reduction in execution time of all the benchmarks with different SPMs sizes for the proposed data transfer strategies, and each bar also shows the time distribution among program phases. Four SPM sizes are studied: 32, 64, 128 and 256 KB with access times of 2, 3, 4 and 6 cycles, respectively. The ROB is augmented to 192 entries in the experiments with 256 KB SPMs to tolerate the latency.

Results show that, for all the data transfer strategies, the average execution time of all the benchmarks decreases as the size of the SPMs increase. The size of the SPMs has a big impact in the runtime phases, that represent more than 15% of the total execution

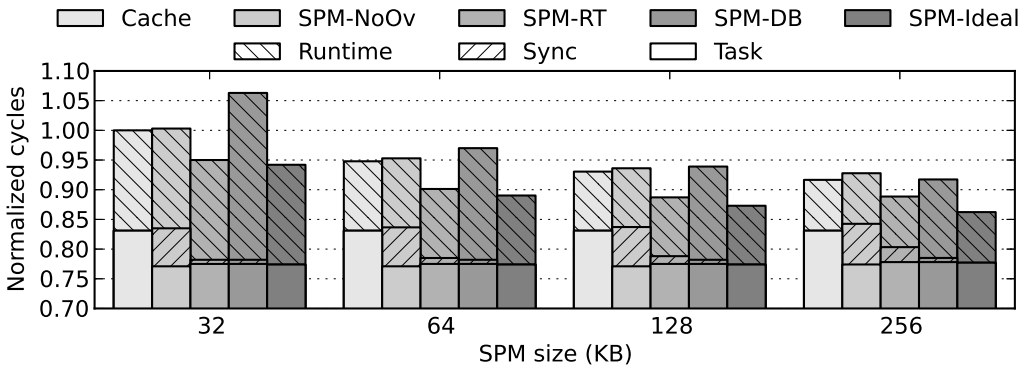

Figure 5.11: Reduction of execution time for different SPM sizes

time with 32 KB SPMs and is reduced to less than 10% with 256 KB SPMs. This happens because bigger SPMs allow to use coarser-grained tasks in 6 of the 10 benchmarks, so less tasks are needed to perform the computation and the runtime overhead is lower. In the rest of benchmarks the task granularity is fixed by the way the benchmark is decomposed in tasks, so having bigger SPM sizes does not decrease the runtime overhead. Another effect of augmenting the size of the SPMs is that the synchronization time increases in the SPM-RT approach for some benchmarks due to the reduced length of the runtime phases and the bigger DMA transfers. This causes that, for SPM-RT, the percentage of time spent in synchronization phases goes from less than 1% in all benchmarks with SPMs of 32 KB to an average 3% with SPMs of 256 KB, reaching up to 8% in `md5`. It can also be observed that the size of the SPMs has a negligible effect on the execution time of the tasks, as the additional latency of bigger SPMs can be hidden with the execution of other instructions. All together, increasing the size of the SPMs from 32 KB to 256 KB provides average execution time reductions of more than 7% in all cases.

Another solution to mitigate the runtime system overheads is to add a hardware runtime system that manages the execution of the tasks [57, 89, 127]. These solutions report speedups of 2 to 3 orders of magnitude for the runtime system phases, effectively eliminating the overheads caused by fine-grained tasks.

Figure 5.12 shows an estimation of the performance benefits provided by the hybrid memory hierarchy when combined with a task runtime system with hardware support. In order to estimate the performance in this kind of systems the execution time of the runtime system phases is accelerated by a factor of 100x. The SPM-RT configuration is not considered in the study because the runtime phases are too short to hide the cost of the DMA transfers.

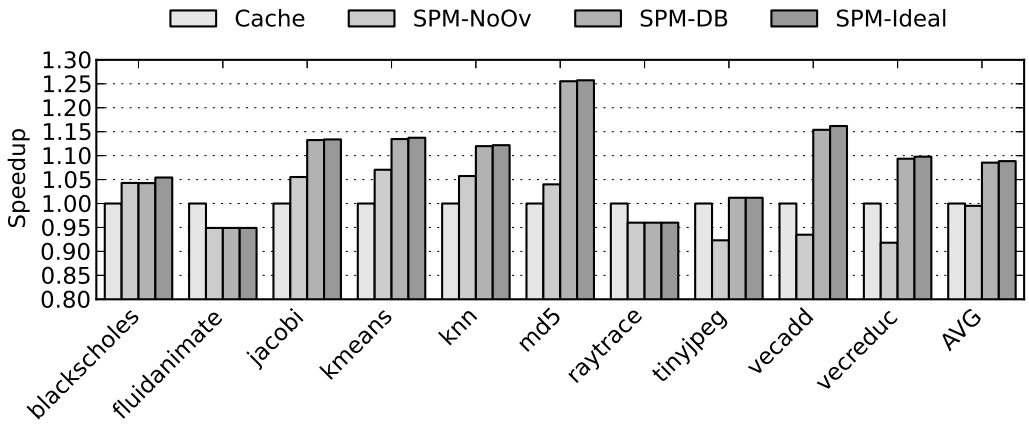

Figure 5.12: Speedup with a hardware runtime system

It can be observed that the results for SPM-NoOv are very similar to the ones presented in Figure 5.7, as the impact in performance of the hardware runtime system is equal for both the baseline cache hierarchy and the SPM-NoOv configuration. In contrast, the hardware runtime system completely eliminates the runtime overhead introduced by the bigger amount of tasks in the SPM-DB approach, and provides close to ideal performance because DMA transfers are completely overlapped with the execution of the tasks. On average, an speedup of 8% is achieved against the cache hierarchy, reaching up to 22% for `md5`. These estimated results indicate that SPM-DB is the appropriate solution for future multicores with hardware support for the runtime system.

5.4 Summary and Concluding Remarks

This chapter proposes to give to the runtime system of task data-flow programming models the responsibility of automatically managing the SPMs in a hybrid memory hierarchy with caches and SPMs. SPMs are more power-efficient than caches and they do not generate coherence traffic but they impose programmability difficulties. In task programming models the task dependences can be exploited to manage the SPMs in the runtime system, transparently to the programmer and in a generic way.

Task data-flow programming models are very well suited for SPMs, since the task dependences specify what data is going to be accessed during the execution of the tasks, and the programming model ensures that no data races will happen on the data dependences during the execution of the tasks. In addition, the DMA transfers can be overlapped with different phases of the execution model. These properties are exploited in this chapter to

map the input and output dependences of the tasks to the SPMs of the hybrid memory hierarchy and to apply optimizations like locality-aware scheduling and overlapping of DMA transfers with computation.

Results show that, in a multicore with 32 cores, the hybrid memory hierarchy outperforms cache hierarchies by up to 5%, consumes up to 27% less energy and reduces NoC traffic by up to 31% when DMA transfers are not overlapped with useful work. Overlapping the DMA transfers further improves performance, achieving speedups of up to 16% when overlapping with the task scheduler. Double buffering with the previous task increases the runtime system overheads, so it is better suited for architectures with hardware runtime systems or SPMs of hundreds of kilobytes.

Chapter 6
Conclusions

This chapter summarizes the main conclusions and contributions of this thesis and presents the future research lines opened by this work. Then it shows the list of publications produced during the realization of this thesis and acknowledges the financial support.

6.1 Goals, Contributions and Main Conclusions

Traditionally, cache-coherent shared memory has been the most commonly used memory organization for chip multiprocessors. The main advantage of this solution is that the architecture can be programmed with shared memory programming models, given that the cache hierarchy is able to move the data and to keep it coherent between all the caches without any intervention from the programmer. Unfortunately, performing these operations in hardware limits the scaling of the number of cores in the architecture due to the power consumption originated in the caches and the amount traffic in the interconnection network needed to maintain coherence.

Combining ScratchPad Memories (SPMs) and caches in a hybrid memory hierarchy is a good solution to alleviate these problems, as SPMs are more power-efficient than caches and they do not generate coherence traffic. However, since SPMs are managed by software, they require the programmer to partition the data, to explicitly program data transfers, and to keep coherence between different copies of the data.

A promising solution to solve the programmability issues of hybrid memory hierarchies is to allow the programmer to use shared memory programming models and to automatically generate code that manages the SPMs, so the programmer does not need to explicitly manage the data. Unfortunately, current compilers and runtime systems encounter serious limitations to automatically generate code for hybrid memory hierarchies from shared memory programming models.

The goal of this thesis is to propose a combination of hardware, compiler and runtime system techniques to manage the SPMs of hybrid memory hierarchies in fork-join and task programming models. The proposed techniques allow to program hybrid memory hierarchies with these two well-known and easy-to-use forms of shared memory programming models, capitalizing on the benefits of hybrid memory hierarchies in terms of power consumption and network traffic without harming the programmability of the architecture.

The first contribution of this thesis is a coherence protocol to automatically manage the SPMs of hybrid memory hierarchies in fork-join programming models. The proposed techniques allow the compiler to always generate code to manage the SPMs with tiling software caches, even in the presence of unknown memory aliasing hazards between memory references to the SPMs and memory references to the cache hierarchy. In a hardware/software coordinated mechanism, the compiler identifies memory references with possible aliasing hazards and generates a special form of memory instruction for them. On the hardware side, a set of directories track what data is mapped to the SPMs and divert these memory accesses with unknown aliases to the correct copy of the data.

The second contribution of this thesis is a set of techniques for runtime systems of task programming models to manage the SPMs of hybrid memory hierarchies. The characteristics of these programming models are exploited by the runtime system to map the data specified in the task dependences to the SPMs of the core where the tasks are executed. The runtime system also applies optimizations to hide the communication costs of the data transfers, overlapping them with either the runtime system phases or the execution of the previous tasks, and schedules tasks to cores aiming to exploit data locality in the SPMs. In addition, the proposed techniques are combined with mechanisms that reduce the impact of fine-grained tasks, such as hardware runtime systems or big SPM sizes.

These two contributions make feasible that the SPMs of hybrid memory hierarchies are managed transparently to the programmer in fork-join and task programming models, achieving the main goal of this thesis. The immediate benefit of this accomplishment is that the advantages in power consumption and network traffic provided by hybrid memory hierarchies can be exploited maintaining the programming simplicity of shared memory programming models.

6.2 Future Work

The proposals presented in this thesis open the door to new research topics that could be explored in the future. Among others, three main research lines can be of great interest.

- *Optimization of cache hierarchies for hybrid memory hierarchies*. This thesis has shown that adding SPMs alongside the cache hierarchy provides important benefits, but this scheme could be further optimized by adapting the configuration of the cache hierarchy. The introduction of the SPMs drastically changes the way the data is accessed and moved in the chip, so the design of the cache hierarchy could be revisited to better fulfill the requirements of this new scenario. In particular, in the hybrid memory hierarchy the strided accesses are served by the SPMs, so some elements of the cache hierarchy such as the L1 D-cache or the L1 prefetcher could be optimized for random accesses. In addition, since most of the data set is moved from the main memory to the SPMs without using the cache hierarchy, the size of the big last level caches that are used nowadays could be scaled down to further reduce the power consumption or to make a better utilization of the available transistors in the chip.

- *Automatic task coarsening in task programming models*. The main problem of automatically managing the SPMs of the hybrid memory hierarchy in task programming models is that the size of the SPMs limits the size of the tasks. As a first approach, this thesis assumes that the code is written in such a way that the size of the task dependences is always smaller than the size of the SPMs. In order to release the programmer from this responsibility, many solutions to perform automatic task coarsening could be explored: a pure static approach where the compiler uses analyses to determine the task size and generates code to fuse or split tasks accordingly, a pure dynamic approach where it is the runtime system that fuses or splits tasks during the execution, a hybrid approach where the compiler and the runtime system cooperate to adapt the task size, or even other techniques such as using a lightweight pagination mechanism on the SPMs.

- *Automatic management of 3D-stacked DRAM memories*. Upcoming architectures like the Intel Knights Landing are going to incorporate 3D-stacked DRAM memories. These memories offer a much higher bandwidth than traditional off-chip main memories, and they are expected to be widely used in the future. However, it is

still unclear how the 3D-stacked DRAM memories will be architected as part of the system. The Intel Knights Landing will offer three possible configurations: (1) giving the 3D-stacked DRAM its own address space and provide an interface so that programmers explicitly manage the memory, (2) using the 3D-stacked DRAM as a hardware-managed cache, and (3) a hybrid scheme that combines the two previous approaches. The configuration that exposes the 3D-stacked DRAM to the programmer in a separate address space raises the same programmability difficulties than integrating SPMs in the memory hierarchy, so the techniques proposed in this thesis could be revisited to automatically manage 3D-stacked DRAM memories in shared memory programming models, transparently to the programmer.

6.3 Publications

The publications derived from the research done during this thesis are listed below. The first list contains the publications directly related to the thesis, while the second list shows the publications from works on other topics done during the realization of this thesis.

6.3.1 Publications of the Thesis

- Lluc Alvarez, Miquel Moreto, Marc Casas, Emilio Castillo, Xavier Martorell, Jesús Labarta, Eduard Ayguadé and Mateo Valero, "Runtime-Guided Management of Scratchpad Memories in Multicore Architectures", to appear in *PACT '15: Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques*, 2015.

- Lluc Alvarez, Lluís Vilanova, Miquel Moreto, Marc Casas, Marc Gonzàlez, Xavier Martorell, Nacho Navarro, Eduard Ayguadé and Mateo Valero, "Coherence Protocol for Transparent Management of Scratchpad Memories in Shared Memory Manycore Architectures", in *ISCA '15: Proceedings of the ACM/IEEE 41st International Symposium on Computer Architecture*, pp. 720-732. IEEE Computer Society, 2015.

- Lluc Alvarez, Lluís Vilanova, Marc Gonzàlez, Xavier Martorell, Nacho Navarro and Eduard Ayguadé, "Hardware-Software Coherence Protocol for the Coexistence of Caches and Local Memories", in *IEEE Transactions on Cumputers*, vol. 64, no. 1, pp. 152-165. IEEE Computer Society, 2015.

- Lluc Alvarez, Lluís Vilanova, Marc Gonzàlez, Xavier Martorell, Nacho Navarro and Eduard Ayguadé, "Hardware-Software Coherence Protocol for the Coexistence of Caches and Local Memories", in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, article no. 89. IEEE Computer Society, 2012.

- Lluc Alvarez, Nikola Vujic, Lluís Vilanova, Ramon Bertran, Marc Gonzàlez, Xavier Martorell, Nacho Navarro and Eduard Ayguadé, "Hardware/Software Coherence in Hybrid Memory Models", Technical Report UPC-DAC-RR-CAP-2011-21, Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya. 2011.

6.3.2 Other Publications

- Marc Casas, Miquel Moreto, Lluc Alvarez, Emilio Castillo, Dimitrios Chasapis, Timothy Hayes, Luc Jaulmes, Oscar Palomar, Osman Unsal, Adrian Cristal, Eduard Ayguadé, Jesús Labarta and Mateo Valero, "Runtime-Aware Architectures", in *Euro-Par 2015: Parallel Processing*, pp. 16-27. Springer Berlin Heidelberg, 2015.

- Nikola Vujic, Lluc Alvarez, Marc Gonzàlez, Xavier Martorell and Eduard Ayguadé, "DMA-Circular: an Enhanced High Level Programmable DMA Controller for Optimized Management of On-chip Local Memories", in *CF '12: Proceedings of the 9th International Conference on Computing Frontiers*, pp. 113-122. ACM, 2012.

- Lluc Alvarez, Ramon Bertran, Marc Gonzàlez, Xavier Martorell, Nacho Navarro and Eduard Ayguadé, "Design Space Exploration For Aggressive Core Replication Schemes in CMPs", in *HPDC '11: Proceedings of the 20th ACM International Symposium on High Performance Distributed Computing*, pp. 269-270. ACM, 2011.

- Nikola Vujic, Lluc Alvarez, Marc Gonzàlez, Xavier Martorell and Eduard Ayguadé, "DMA-based Programmable Caches For On-chip Local Memories", Technical Report UPC-DAC-RR-CAP-2011-20, Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya. 2011.

- Lluc Alvarez, Ramon Bertran, Marc Gonzàlez, Xavier Martorell, Nacho Navarro and Eduard Ayguadé, "Design Space Exploration of CMPs with Caches and Local Memories", Technical Report UPC-DAC-RR-CAP-2011-19, Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya. 2011.

- Julio Merino-Vidal, Lluc Alvarez, Nacho Navarro and Marisa Gil, "Cetra: A Trace and Analysis Framework for the Evaluation of Cell BE Systems", in *ISPASS '09: Proceedings of the 2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 43-52. IEEE Computer Society, 2009.

- Nikola Vujic, Lluc Alvarez, Marc Gonzàlez Tallada, Xavier Martorell and Eduard Ayguadé, "Adaptive and Speculative Memory Consistency Support for Multi-Core Architectures with On-Chip Local Memories", in *LCPC '09: Proceedings of the 22nd International Workshop on Languages and Compilers for Parallel Computing*, pp 218-232. Springer Berlin Heidelberg, 2009.

- Lluc Alvarez, Marc Gonzàlez, Marisa Gil, Nacho Navarro, Xavier Martorell and Eduard Ayguadé, "Thread-Level Speculation in Heterogenous Multicore Architectures", in *ACACES 2009 Poster Abstracts: Advanced Computer Architecture and Compilation for Embedded Systems*. 2009.

- Julio M. Merino-Vidal, Lluc Alvarez, Marisa Gil and Nacho Navarro, "An introduction to Cetra: A tool-set for the evaluation of Cell systems", in *ACACES 2008 Poster Abstracts: Advanced Computer Architecture and Compilation for Embedded Systems*. 2008.

- Marisa Gil, Lluc Alvarez, Xavier Joglar, Judit Planas and Xavier Martorell, "Operating System Support For Heterogeneous Multicore Architectures", Technical Report UPC-DAC-RR-CAP-2007-40, Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya. 2007.

6.4 Financial Support

This thesis has been financially supported by the Spanish Government (grant SEV-2011-00067 of the Severo Ochoa Program), by the Spanish Ministry of Science and Innovation (contracts TIN2007-60625, TIN2012-34557 and CSD2007-00050), by the Generalitat de

Catalunya (contracts 2009-SGR-980, 2014-SGR-1051 and 2014-SGR-1272), by the RoMoL ERC Advanced Grant (GA 321253), by the HiPEAC Network of Excellence (contracts EU FP7/ICT 217068 and 287759), and by the BSC-IBM collaboration agreement.

Bibliography

[1] S. V. Adve and H.-J. Boehm. Memory Models: A Case for Rethinking Parallel Languages and Hardware. *Communications of the ACM*, 53(8):90–101, Aug. 2010.

[2] A. Agarwal, H. Li, and K. Roy. DRG-cache: A Data Retention Gated-ground Cache for Low Power. In *Proceedings of the 39th Annual Design Automation Conference*, DAC '02, pages 473–478, 2002. ACM.

[3] D. H. Albonesi. Selective Cache Ways: On-demand Cache Resource Allocation. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 32, pages 248–259, 1999. IEEE Computer Society.

[4] M. Alisafaee. Spatiotemporal Coherence Tracking. In *Proceedings of the 45th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 45, pages 341–350, 2012. IEEE Computer Society.

[5] F. Angiolini, L. Benini, and A. Caprara. Polynomial-time Algorithm for On-chip Scratchpad Memory Partitioning. In *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '03, pages 318–326, 2003. ACM.

[6] F. Angiolini, F. Menichelli, A. Ferrero, L. Benini, and M. Olivieri. A Post-compiler Approach to Scratchpad Mapping of Code. In *Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '04, pages 259–267, 2004. ACM.

[7] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, K. A. Yelick, M. J. Demmel, W. Plishker, J. Shalf, S. Williams, and K. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, Electri-

cal Engineering and Computer Sciences Department, University of California at Berkeley, 2009.

[8] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Proceedings of the 15st International Conference on Parallel and Distributed Computing*, Euro-Par 2009, pages 863–874, 2009. Springer-Verlag.

[9] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The Design of OpenMP Tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, Mar. 2009.

[10] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS Parallel Benchmarks. *International Journal of High Performance Computer Applications*, 5(3):63–73, Sept. 1991.

[11] J. Balart, A. Duran, M. Gonzàlez, X. Martorell, E. Ayguadé, and J. Labarta. Nanos Mercurium: a Research Compiler for OpenMP. In *Proceedings of the 6th European Workshop on OpenMP*, EWOMP '04, pages 103–109, 2004.

[12] J. Balart, M. González, X. Martorell, E. Ayguadé, Z. Sura, T. Chen, T. Zhang, K. O'Brien, and K. M. O'Brien. A Novel Asynchronous Software Cache Implementation for the Cell-BE Processor. In *Proceedings of the 20th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '07, pages 125–140. Springer-Verlag, 2007.

[13] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory Hierarchy Reconfiguration for Energy and Performance in General-purpose Processor Architectures. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 33, pages 245–257, 2000. IEEE Computer Society.

[14] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad Memory: Design Alternative for Cache On-chip Memory in Embedded Systems. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, CODES '02, pages 73–78, 2002. ACM.

[15] B. Batson and T. N. Vijaykumar. Reactive-Associative Caches. In *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques*, PACT '01, pages 49–60, 2001. IEEE Computer Society.

[16] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing Locality and Independence with Logical Regions. In *Proceedings of the 2012 ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 66:1–66:11, 2012. IEEE Computer Society.

[17] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta. CellSs: A Programming Model for the Cell BE Architecture. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, pages 86:1–86:11, 2006. ACM.

[18] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta. Exploiting Locality on the Cell/B.E. Through Bypassing. In *Proceedings of the 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*, SAMOS '09, pages 318–328, 2009. Springer-Verlag.

[19] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta. Just-in-Time Renaming and Lazy Write-Back on the Cell/B.E. In *Proceedings of the 2009 International Conference on Parallel Processing Workshops*, ICPPW '09, pages 138–145, 2009. IEEE Computer Society.

[20] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta. A Study of Speculative Distributed Scheduling on the Cell/B.E. In *Proceedings of the 2011 IEEE 25th International Parallel and Distributed Processing Symposium*, IPDPS '11, pages 140–151, 2011. IEEE Computer Society.

[21] T. B. Berg. Maintaining I/O Data Coherence in Embedded Multicore Systems. *IEEE Micro*, 29(3):10–19, May 2009.

[22] R. Bertran, M. Gonzàlez, X. Martorell, N. Navarro, and E. Ayguadé. Local Memory Design Space Exploration for High-Performance Computing. *The Computer Journal*, 54(5):786–799, May 2011.

[23] J. Beveridge and B. Wiener. *Multithreading Applications in Win32: The Complete Guide to Threads*. Addison-Wesley Longman Publishing, 1997.

[24] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 72–81, 2008. ACM.

[25] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The Gem5 Simulator. *SIGARCH Computer Architure News*, 39(2):1–7, Aug. 2011.

[26] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '95, pages 207–216, 1995. ACM.

[27] M. Boettcher, G. Gabrielli, B. M. Al-Hashimi, and D. Kershaw. MALEC: A Multiple Access Low Energy Cache. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 368–373, 2013. IEEE Computer Society.

[28] J. Bueno, X. Martorell, R. M. Badia, E. Ayguadé, and J. Labarta. Implementing OmpSs Support for Regions of Data in Architectures with Multiple Address Spaces. In *Proceedings of the 27th ACM International Conference on Supercomputing*, ICS '13, pages 359–368, 2013. ACM.

[29] D. R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley Longman Publishing, 1997.

[30] B. Calder, D. Grunwald, and J. Emer. Predictive Sequential Associative Cache. In *Proceedings of the 1996 IEEE 2nd International Symposium on High-Performance Computer Architecture*, HPCA '96, pages 244–253, 1996. IEEE Computer Society.

[31] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Improving Multiprocessor Performance with Coarse-Grain Coherence Tracking. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, ISCA '05, pages 246–257, 2005. ACM.

[32] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Improving Multiprocessor Performance with Coarse-Grain Coherence Tracking. *SIGARCH Computer Architecture News*, 33(2):246–257, May 2005.

[33] N. P. Carter, A. Agrawal, S. Borkar, R. Cledat, H. David, D. Dunning, J. Fryman, I. Ganev, R. A. Golliver, R. Knauerhase, R. Lethin, B. Meister, A. K. Mishra, W. R. Pinfold, J. Teller, J. Torrellas, N. Vasilache, G. Venkatesh, and J. Xu. Runnemede: An Architecture for Ubiquitous High-Performance Computing. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture*, HPCA '13, pages 198–209, 2013. IEEE Computer Society.

[34] M. Casas, M. Moreto, L. Alvarez, E. Castillo, D. Chasapis, T. Hayes, L. Jaulmes, O. Palomar, O. Unsal, A. Cristal, et al. Runtime-Aware Architectures. In *Proceedings of the 21st International Conference on Parallel and Distributed Computing*, Euro-Par 2015, pages 16–27, 2015. Springer-Verlag.

[35] L. Chen, L. Liu, S. Tang, L. Huang, Z. Jing, S. Xu, D. Zhang, and B. Shou. Unified Parallel C for GPU Clusters: Language Extensions and Compiler Implementation. In *Proceedings of the 23rd International Workshop on Languages and Compilers for Parallel Computing*, LCPC '10, pages 151–165, 2011. Springer-Verlag.

[36] T. Chen, Z. Sura, K. M. O'Brien, and J. K. O'Brien. Optimizing the Use of Static Buffers for DMA on a CELL Chip. In *Proceedings of the 19th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '06, pages 314–329. Springer-Verlag, 2006.

[37] H. Cho, B. Egger, J. Lee, and H. Shin. Dynamic Data Scratchpad Memory Management for a Memory Subsystem with an MMU. In *Proceedings of the 2007 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '07, pages 195–206, 2007. ACM.

[38] H. Cook, K. Asanovic, and D. A. Patterson. Virtual Local Stores: Enabling Software-Managed Memory Hierarchies in Mainstream Computing Environments. Technical Report UCB/EECS-2009-131, Electrical Engineering and Computer Sciences Department, University of California at Berkeley, 2009.

[39] B. Cuesta, A. Ros, M. E. Gomez, A. Robles, and J. Duato. Increasing the Effectiveness of Directory Caches by Avoiding the Tracking of Noncoherent Memory Blocks. *IEEE Transactions on Computers*, 62(3):482–495, Mar. 2013.

[40] B. A. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. F. Duato. Increasing the Effectiveness of Directory Caches by Deactivating Coherence for Private Memory Blocks. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 93–104, 2011. ACM.

[41] B. A. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. F. Duato. Increasing the Effectiveness of Directory Caches by Deactivating Coherence for Private Memory Blocks. *SIGARCH Computer Architecture News*, 39(3):93–104, June 2011.

[42] W. J. Dally, F. Labonte, A. Das, P. Hanrahan, J.-H. Ahn, J. Gummaraju, M. Erez, N. Jayasena, I. Buck, T. J. Knight, and U. J. Kapasi. Merrimac: Supercomputing with Streams. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, SC '03, pages 35–42, 2003. ACM.

[43] A. Das, W. J. Dally, and P. Mattson. Compiling for Stream Processing. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, PACT '06, pages 33–42, 2006. ACM.

[44] R. H. Dennard, F. H. Gaensslen, H. nien Yu, V. L. Rideout, E. Bassous, Andre, and R. Leblanc. Design of Ion-implanted MOSFETs with Very Small Physical Dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, Oct. 1974.

[45] A. Deutsch. Interprocedural May-alias Analysis for Pointers: Beyond K-limiting. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 230–241, 1994. ACM.

[46] A. Deutsch. Interprocedural May-alias Analysis for Pointers: Beyond K-limiting. *ACM SIGPLAN Notices*, 29(6):230–241, June 1994.

[47] A. S. Dhodapkar and J. E. Smith. Managing Multi-configuration Hardware via Dynamic Working Set Analysis. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ISCA '02, pages 233–244, 2002. IEEE Computer Society.

[48] A. S. Dhodapkar and J. E. Smith. Managing Multi-configuration Hardware via Dynamic Working Set Analysis. *SIGARCH Computer Architure News*, 30(2):233–244, May 2002.

[49] R. Dolbeau, S. Bihan, and F. Bodin. HMPP: A hybrid multi-core parallel programming environment. In *Proceedings of the 1st Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU-1, 2007. ACM.

[50] C. Dubach, P. Cheng, R. Rabbah, D. F. Bacon, and S. J. Fink. Compiling a High-level Language for GPUs: (via Language Support for Architectures and Compilers). In *Proceedings of the ACM SIGPLAN 2012 Conference on Programming Language Design and Implementation*, PLDI '12, pages 1–12, 2012. ACM.

[51] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. OmpSs: A Proposal for Programming Heterogeneous Multi-core Architectures. *Parallel Processing Letters*, 21(2):173–193, 2011.

[52] H. C. Edwards and C. R. Trott. Kokkos: Enabling Performance Portability Across Manycore Architectures. In *Proceedings of the 2013 Extreme Scaling Workshop*, XSW '13, pages 18–24, 2013. IEEE Computer Society.

[53] B. Egger, C. Kim, C. Jang, Y. Nam, J. Lee, and S. L. Min. A Dynamic Code Placement Technique for Scratchpad Memory Using Postpass Optimization. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, pages 223–233, 2006. ACM.

[54] B. Egger, J. Lee, and H. Shin. Scratchpad Memory Management for Portable Systems with a Memory Management Unit. In *Proceedings of the 6th ACM/IEEE International Conference on Embedded Software*, EMSOFT '06, pages 321–330, 2006. ACM.

[55] A. E. Eichenberger, J. K. O'Brien, K. M. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. K. Gschwind, R. Archambault, Y. Gao, and R. Koo. Using Advanced Compiler Technology to Exploit the Performance of the Cell Broadband Engine™Architecture. *IBM Systems Journal*, 45(1):59–84, Jan. 2006.

[56] A. E. Eichenberger, K. O'Brien, K. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, and

M. Gschwind. Optimizing Compiler for the CELL Processor. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, PACT '05, pages 161–172, 2005. IEEE Computer Society.

[57] Y. Etsion, F. Cabarcas, A. Rico, A. Ramirez, R. M. Badia, E. Ayguade, J. Labarta, and M. Valero. Task Superscalar: An Out-of-Order Task Pipeline. In *Proceedings of the 43rd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 43, pages 89–100, 2010. IEEE Computer Society.

[58] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the Memory Hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, pages 83:1–83:11, 2006. ACM.

[59] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy Caches: Simple Techniques for Reducing Leakage Power. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ISCA '02, pages 148–157, 2002. IEEE Computer Society.

[60] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy Caches: Simple Techniques for Reducing Leakage Power. *SIGARCH Computer Architecture News*, 30(2):148–157, May 2002.

[61] V. Garcia, A. Rico, C. Villavieja, P. Carpenter, N. Navarro, and A. Ramirez. Adaptive Runtime-Assisted Block Prefetching on Chip-Multiprocessors. In *Proceedings of the 3rd International Workshop on On-chip Memory Hierarchies and Interconnets*, OMHI '14, pages 1888–1892, 2014. Springer-Verlag.

[62] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W.-m. W. Hwu. An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 347–358, 2010. ACM.

[63] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W.-m. W. Hwu. An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems. *SIGARCH Computer Architectures News*, 38(1):347–358, Mar. 2010.

[64] M. Ghosh, E. Özer, S. Biles, and H.-H. S. Lee. Efficient System-on-chip Energy Management with a Segmented Bloom Filter. In *Proceedings of the 19th International Conference on Architecture of Computing Systems*, ARCS '06, pages 283–297, 2006. Springer-Verlag.

[65] M. Ghosh, E. Ozer, S. Ford, S. Biles, and H.-H. S. Lee. Way Guard: A Segmented Counting Bloom Filter Approach to Reducing Energy for Set-associative Caches. In *Proceedings of the 2009 International Symposium on Low Power Electronics and Design*, ISLPED '09, pages 165–170, 2009. ACM.

[66] Peter N. Glaskowsky. NVIDIA's Fermi: The First Complete GPU Computing Architecture. White paper. 2009.

[67] M. Gonzàlez, N. Vujic, X. Martorell, E. Ayguadé, A. E. Eichenberger, T. Chen, Z. Sura, T. Zhang, K. O'Brien, and K. O'Brien. Hybrid Access-specific Software Cache Techniques for the Cell BE Architecture. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 292–302, 2008. ACM.

[68] K. Gregory and A. Miller. *C++ AMP: Accelerated Massive Parallelism with Microsoft Visual C++*. Developer Reference. Microsoft Press, 2012.

[69] J. Gummaraju, M. Erez, J. Coburn, M. Rosenblum, and W. J. Dally. Architectural Support for the Stream Execution Model on General-Purpose Processors. In *Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques*, PACT '07, pages 3–12, 2007. IEEE Computer Society.

[70] J. Gummaraju and M. Rosenblum. Stream Programming on General-Purpose Processors. In *Proceedings of the 38th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 38, pages 343–354, 2005. IEEE Computer Society.

[71] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive NUCA: Near-optimal Block Placement and Replication in Distributed Caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 184–195, 2009. ACM.

[72] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive NUCA: Near-optimal Block Placement and Replication in Distributed Caches. *SIGARCH Computer Architecture News*, 37(3):184–195, June 2009.

[73] L. J. Hendren, J. Hummell, and A. Nicolau. Abstractions for Recursive Pointer Data Structures: Improving the Analysis and Transformation of Imperative Programs. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, pages 249–260, 1992. ACM.

[74] L. J. Hendren, J. Hummell, and A. Nicolau. Abstractions for Recursive Pointer Data Structures: Improving the Analysis and Transformation of Imperative Programs. *ACM SIGPLAN Notices*, 27(7):249–260, July 1992.

[75] K. Inoue, T. Ishihara, and K. Murakami. Way-predicting Set-associative Cache for High Performance and Low Energy Consumption. In *Proceedings of the 1999 International Symposium on Low Power Electronics and Design*, ISLPED '99, pages 273–275, 1999. ACM.

[76] Intel 64 and IA-32 Architectures Software Developer's Manual. January 2011.

[77] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August. Automatic CPU-GPU Communication Management and Optimization. In *Proceedings of the ACM SIGPLAN 2011 Conference on Programming Language Design and Implementation*, PLDI '11, pages 142–151, 2011. ACM.

[78] A. Janapsatya, A. Ignjatović, and S. Parameswaran. A Novel Instruction Scratchpad Memory Optimization Method Based on Concomitance Metric. In *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, ASP-DAC '06, pages 612–617, 2006. IEEE Computer Society.

[79] J. Kahle. The Cell Processor Architecture. In *Proceedings of the 38th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 38, page 3, 2005. IEEE Computer Society.

[80] L. V. Kale and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '93, pages 91–108, 1993. ACM.

[81] M. Kandemir and A. Choudhary. Compiler-directed Scratch Pad Memory Hierarchy Design and Management. In *Proceedings of the 39th Annual Design Automation Conference*, DAC '02, pages 628–633, 2002. ACM.

[82] M. Kandemir, J. Ramanujam, J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. Dynamic Management of Scratch-pad Memory Space. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, pages 690–695, 2001. ACM.

[83] S. Kaxiras, Z. Hu, and M. Martonosi. Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, ISCA '01, pages 240–251, 2001. ACM.

[84] S. Kaxiras, Z. Hu, and M. Martonosi. Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power. *SIGARCH Computer Architecture News*, 29(2):240–251, May 2001.

[85] B. Khailany, W. J. Dally, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, A. Chang, and S. Rixner. Imagine: Media Processing with Streams. *IEEE Micro*, 21(2):35–46, Mar. 2001.

[86] D. Kim, J. Ahn, J. Kim, and J. Huh. Subspace Snooping: Filtering Snoops with Operating System Support. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 111–122, 2010. ACM.

[87] N. S. Kim, K. Flautner, D. Blaauw, and T. Mudge. Drowsy Instruction Caches: Leakage Power Reduction Using Dynamic Voltage Scaling and Cache Sub-bank Prediction. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 35, pages 219–230, 2002. IEEE Computer Society.

[88] M. Kistler, M. Perrone, and F. Petrini. Cell Multiprocessor Communication Network: Built for Speed. *IEEE Micro*, 26(3):10–23, May 2006.

[89] S. Kumar, C. J. Hughes, and A. Nguyen. Carbon: Architectural Support for Fine-grained Parallelism on Chip Multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 162–173, 2007. ACM.

[90] W. Landi and B. G. Ryder. A Safe Approximate Algorithm for Interprocedural Aliasing. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, pages 235–248, 1992. ACM.

[91] W. Landi and B. G. Ryder. A Safe Approximate Algorithm for Interprocedural Aliasing. *ACM SIGPLAN Notices*, 27(7):235–248, July 1992.

[92] J. Lee, S. Seo, C. Kim, J. Kim, P. Chun, Z. Sura, J. Kim, and S. Han. COMIC: A Coherent Shared Memory Interface for Cell BE. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 303–314, 2008. ACM.

[93] S. Lee and R. Eigenmann. OpenMPC: Extended OpenMP Programming and Tuning for GPUs. In *Proceedings of the 2010 ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, 2010. IEEE Computer Society.

[94] S. Lee, S.-J. Min, and R. Eigenmann. OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 101–110, 2009. ACM.

[95] L. Li, L. Gao, and J. Xue. Memory Coloring: A Compiler Approach for Scratchpad Memory Management. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, PACT '05, pages 329–338, 2005. IEEE Computer Society.

[96] L. Li, I. Kadayif, Y.-F. Tsai, N. Vijaykrishnan, M. T. Kandemir, M. J. Irwin, and A. Sivasubramaniam. Leakage Energy Management in Cache Hierarchies. In *Proceedings of the 11th International Conference on Parallel Architectures and Compilation Techniques*, PACT '02, pages 131–140, 2002. ACM.

[97] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proceedings of the 42nd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 42, pages 469–480, 2009. IEEE Computer Society.

[98] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz. Smart Memories: A Modular Reconfigurable Architecture. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 161–171, 2000. IEEE Computer Society.

[99] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz. Smart Memories: A Modular Reconfigurable Architecture. *SIGARCH Computer Architecture News*, 28(2):161–171, May 2000.

[100] M. Manivannan, A. Negi, and P. Stenström. Efficient Forwarding of Producer-Consumer Data in Task-Based Programs. In *Proceedings of the 2013 42nd International Conference on Parallel Processing*, ICPP '13, pages 517–522, 2013. IEEE Computer Society.

[101] M. Manivannan and P. Stenstrom. Runtime-Guided Cache Coherence Optimizations in Multi-core Architectures. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 625–636, 2014. IEEE Computer Society.

[102] R. Min, W.-B. Jone, and Y. Hu. Location Cache: A Low-power L2 Cache System. In *Proceedings of the 2004 International Symposium on Low Power Electronics and Design*, ISLPED '04, pages 120–125, 2004. ACM.

[103] A. Moshovos. RegionScout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, ISCA '05, pages 234–245, 2005. ACM.

[104] A. Moshovos. RegionScout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence. *SIGARCH Computer Architecture News*, 33(2):234–245, May 2005.

[105] T. Mudge. Power: A First-Class Architectural Design Constraint. *IEEE Computer Journal*, 34(4):52–58, Apr. 2001.

[106] F. Mueller. A Library Implementation of POSIX Threads under UNIX. In *Proceedings of the Usenix Winter 1993 Technical Conference*, USENIX '93, pages 29–42. USENIX Association, 1993.

[107] R. Murphy and P. Kogge. On the Memory Access Patterns of Supercomputer Applications: Benchmark Selection and Its Implications. *IEEE Transactions on Computers*, 56(7):937–945, July 2007.

[108] N. Nguyen, A. Dominguez, and R. Barua. Memory Allocation for Embedded Systems with a Compile-time-unknown Scratch-pad Size. In *Proceedings of the 2005*

International Conference on Compilers, Architecture and Synthesis for Embedded Systems, CASES '05, pages 115–125, 2005. ACM.

[109] D. Nicolaescu, A. Veidenbaum, and A. Nicolau. Reducing Power Consumption for High-Associativity Data Caches in Embedded Processors. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '03, pages 1064–1069, 2003. IEEE Computer Society.

[110] NVIDIA CUDA C Programming Guide. Version 7.0. March 2015.

[111] S. Oaks and H. Wong. *Java threads*. O'Reilly Media, 2004.

[112] K. O'Brien, K. O'Brien, Z. Sura, T. Chen, and T. Zhang. Supporting OpenMP on Cell. *International Journal of Parallel Programming*, 36(3):289–311, June 2008.

[113] The OpenACC Application Program Interface. Version 1.0. Novembre 2011.

[114] OpenMP Application Program Interface. Version 3.0. May 2008.

[115] OpenMP Application Program Interface. Version 4.0. July 2013.

[116] Y. Paek, J. Hoeflinger, and D. Padua. Efficient and Precise Array Access Analysis. *ACM Transactions on Programming Language and Systems*, 24(1):65–109, Jan. 2002.

[117] P. R. Panda, N. D. Dutt, and A. Nicolau. Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications. In *Proceedings of the 1997 European Conference on Design and Test*, EDTC '97, pages 7–11, 1997. IEEE Computer Society.

[118] P. R. Panda, N. D. Dutt, and A. Nicolau. On-chip vs. Off-chip Memory: The Data Partitioning Problem in Embedded Processor-based Systems. *ACM Transactions on Design Automation of Electronic Systems*, 5(3):682–704, July 2000.

[119] V. Papaefstathiou, M. G. Katevenis, D. S. Nikolopoulos, and D. Pnevmatikatos. Prefetching and Cache Management Using Task Lifetimes. In *Proceedings of the 27th ACM International Conference on Supercomputing*, ICS '13, pages 325–334, 2013. ACM.

[120] J. P. Perez, P. Bellens, R. M. Badia, and J. Labarta. CellSs: Making It Easier to Program the Cell Broadband Engine Processor. *IBM Journal of Research and Development*, 51(5):593–604, Sept. 2007.

[121] J. Planas, R. M. Badia, E. Ayguade, and J. Labarta. Self-Adaptive OmpSs Tasks in Heterogeneous Environments. In *Proceedings of the 2013 IEEE 27th International Parallel and Distributed Processing Symposium*, IPDPS '13, pages 138–149, 2013. IEEE Computer Society.

[122] M. Powell, S.-H. Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar. Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-submicron Cache Memories. In *Proceedings of the 2000 International Symposium on Low Power Electronics and Design*, ISLPED '00, pages 90–95, 2000. ACM.

[123] M. D. Powell, A. Agarwal, T. N. Vijaykumar, B. Falsafi, and K. Roy. Reducing Set-associative Cache Energy via Way-prediction and Selective Direct-mapping. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 34, pages 54–65, 2001. IEEE Computer Society.

[124] Power ISA. Version 2.06 Revision B. July 2010.

[125] A. Ramirez, F. Cabarcas, B. Juurlink, M. Alvarez Mesa, F. Sanchez, A. Azevedo, C. Meenderinck, C. Ciobanu, S. Isaza, and G. Gaydadjiev. The SARC Architecture. *IEEE Micro*, 30(5):16–29, Sept. 2010.

[126] J. Reinders. *Intel threading building blocks - outfitting C++ for multi-core processor parallelism*. O'Reilly Media, 2007.

[127] D. Sanchez, R. M. Yoo, and C. Kozyrakis. Flexible Architectural Support for Fine-grain Scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 311–322, 2010. ACM.

[128] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA '03, pages 422–433, 2003. ACM.

[129] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture. *SIGARCH Computer Architecture News*, 31(2):422–433, May 2003.

[130] A. Sembrant, E. Hagersten, and D. Black-Schaffer. The Direct-to-Data (D2D) Cache: Navigating the Cache Hierarchy with a Single Lookup. In *Proceedings of the 41st Annual International Symposium on Computer Architecture*, ISCA '14, pages 133–144, 2014. IEEE Computer Society.

[131] A. Sembrant, E. Hagersten, and D. Black-Schaffer. The Direct-to-Data (D2D) Cache: Navigating the Cache Hierarchy with a Single Lookup. *SIGARCH Computer Architecture News*, 42(3):133–144, June 2014.

[132] A. Sembrant, E. Hagersten, and D. Black-Shaffer. TLC: A Tag-less Cache for Reducing Dynamic First Level Cache Energy. In *Proceedings of the 46th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 46, pages 49–61, 2013. IEEE Computer Society.

[133] S. Seo, J. Lee, and Z. Sura. Design and Implementation of Software-Managed Caches for Multicores with Local Memory. In *Proceedings of the 2009 IEEE 15th International Symposium on High-Performance Computer Architecture*, HPCA '09, pages 55–66, 2009. IEEE Computer Society.

[134] J. Shirako, J. M. Zhao, V. K. Nandivada, and V. N. Sarkar. Chunking Parallel Loops in the Presence of Synchronization. In *Proceedings of the 23rd ACM International Conference on Supercomputing*, ICS '09, pages 181–192, 2009. ACM.

[135] A. Sidelnik, S. Maleki, B. L. Chamberlain, M. J. Garzarán, and D. Padua. Performance Portability with the Chapel Language. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, IPDPS '12, pages 582–594, 2012. IEEE Computer Society.

[136] A. Sodani. Race to Exascale: Opportunities and Challenges, Keynote in MICRO 2011.

[137] S. Steinke, N. Grunwald, L. Wehmeyer, R. Banakar, M. Balakrishnan, and P. Marwedel. Reducing Energy Consumption by Dynamic Copying of Instructions Onto Onchip Memory. In *Proceedings of the 15th International Symposium on System Synthesis*, ISSS '02, pages 213–218, 2002. ACM.

[138] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *IEEE Design and Test Journal*, 12(3):66–73, May 2010.

[139] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing, 4th edition, 2013.

[140] Sun Microsystems. Multithreading in the Solaris Operating Environment. A Technical White Paper. 2002.

[141] S. Thoziyoor, J. H. Ahn, M. Monchiero, J. B. Brockman, and N. P. Jouppi. A Comprehensive Memory Modeling Tool and Its Application to the Design and Analysis of Future Memory Hierarchies. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 51–62, 2008. IEEE Computer Society.

[142] S. Thoziyoor, J. H. Ahn, M. Monchiero, J. B. Brockman, and N. P. Jouppi. A Comprehensive Memory Modeling Tool and Its Application to the Design and Analysis of Future Memory Hierarchies. *SIGARCH Computer Architecture News*, 36(3):51–62, June 2008.

[143] V. Tipparaju and J. S. Vetter. GA-GPU: Extending a Library-based Global Address Space Programming Model for Scalable Heterogeneous Computing Systems. In *Proceedings of the 9th Conference on Computing Frontiers*, CF '12, pages 53–64, 2012. ACM.

[144] E. Totoni, J. Torrellas, and L. V. Kale. Using an Adaptive HPC Runtime System to Reconfigure the Cache Hierarchy. In *Proceedings of the 2014 ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 1047–1058, 2014. IEEE Computer Society.

[145] S. Udayakumaran and R. Barua. Compiler-decided Dynamic Memory Allocation for Scratch-pad Based Embedded Systems. In *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '03, pages 276–286, 2003. ACM.

[146] M. Valero, M. Moreto, M. Casas, E. Ayguade, and J. Labarta. Runtime-Aware Architectures: A First Approach. *International Journal on Supercomputing Frontiers and Innovations*, 1(1):29–44, June 2014.

[147] J. J. Valls, A. Ros, J. Sahuquillo, and M. E. Gómez. PS-cache: An Energy-efficient Cache Design for Chip Multiprocessors. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 407–408, 2013. IEEE Computer Society.

[148] J. J. Valls, A. Ros, J. Sahuquillo, and M. E. Gomez. PS-Cache: An Energy-efficient Cache Design for Chip Multiprocessors. *The Journal of Supercomputing*, 71(1):67–86, Jan. 2015.

[149] J. J. Valls, J. Sahuquillo, A. Ros, and M. E. Gomez. The Tag Filter Cache: An Energy-Efficient Approach. In *Proceedings of the 23rd Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, PDP '15, pages 182–189, March 2015. IEEE Computer Society.

[150] H. Vandierendonck, P. Pratikakis, and D. S. Nikolopoulos. Parallel Programming of General-purpose Programs Using Task-based Programming Models. In *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism*, HotPar '11, pages 13–13, 2011. USENIX Association.

[151] M. Verma, L. Wehmeyer, and P. Marwedel. Cache-Aware Scratchpad Allocation Algorithm. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '04, pages 1264–1269, 2004. IEEE Computer Society.

[152] J. Weinberg, M. O. McCracken, E. Strohmaier, and A. Snavely. Quantifying Locality In The Memory Access Patterns of HPC Applications. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05, page 50, 2005. ACM.

[153] R. P. Wilson and M. S. Lam. Efficient Context-sensitive Pointer Analysis for C Programs. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, pages 1–12, 1995. ACM.

[154] R. P. Wilson and M. S. Lam. Efficient Context-sensitive Pointer Analysis for C Programs. *ACM SIGPLAN Notices*, 30(6):1–12, June 1995.

[155] M. Wolfe. Implementing the PGI Accelerator Model. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU-3, pages 43–50, 2010. ACM.

[156] W. A. Wulf and S. A. McKee. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Computer Architure News*, 23(1):20–24, Mar. 1995.

[157] C. Zhang, F. Vahid, and W. Najjar. A Highly Configurable Cache Architecture for Embedded Systems. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA '03, pages 136–146, 2003. ACM.

[158] C. Zhang, F. Vahid, and W. Najjar. A Highly Configurable Cache Architecture for Embedded Systems. *SIGARCH Computer Architure News*, 31(2):136–146, May 2003.

[159] C. Zhang, F. Vahid, J. Yang, and W. Najjar. A Way-halting Cache for Low-energy High-performance Systems. *ACM Transactions on Architecture Code Optimization*, 2(1):34–54, Mar. 2005.

[160] W. Zhang, J. S. Hu, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Compiler-directed Instruction Cache Leakage Optimization. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 35, pages 208–218, 2002. IEEE Computer Society.

[161] Z. Zheng, Z. Wang, and M. Lipasti. Tag Check Elision. In *Proceedings of the 2014 International Symposium on Low Power Electronics and Design*, ISLPED '14, pages 351–356, 2014. ACM.

[162] H. Zhou, M. C. Toburen, E. Rotenberg, and T. M. Conte. Adaptive Mode Control: A Static-Power-Efficient Cache Design. In *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques*, PACT '01, pages 61–70, 2001. IEEE Computer Society.

[163] H. Zhou, M. C. Toburen, E. Rotenberg, and T. M. Conte. Adaptive Mode Control: A Static-power-efficient Cache Design. *ACM Transactions on Embedded Computing Systems*, 2(3):347–372, Aug. 2003.

[164] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao. Using a "Codelet" Program Execution Model for Exascale Machines: Position Paper. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, pages 64–69, 2011. ACM.

List of Figures

List of Tables

Glossary

ALU Arithmetic Logic Unit

API Application Programming Interface

CAM Content-Addressable Memory

CPI Cycles Per Instruction

CPU Central Processing Unit

DMA Direct Memory Access

DRAM Dynamic Random-Access Memory

EIB Element Interconnection Bus

GM Global Memory

GPGPU General-Purpose Graphics Processing Unit

HPC High Performance Computing

ILP Instruction Level Parallelism

LSQ Load/Store Queue

MMU Memory Management Unit

MSHR Miss Status Handling Registers

NoC Network on-Chip

NUCA Non-Uniform Cache Architecture

OS Operating System

PPE Power Processor Element

RAM Random-Access Memory

SIMD Single Instruction Multiple Data

SM Streaming Multiprocessor

SMT Simultaneous MultiThreading

SPE Synergistic Processor Element

SPM ScratchPad Memory

SRAM Static Random-Access Memory

SRF Stream Register File

TDG Task Dependence Graph

TLB Translation Lookaside Buffer

TLP Task Level Parallelism

VLIW Very Long Instruction Word

WaR Write after Read

WaW Write after Write