Hardware-Software Coherence Protocol for the Coexistence of Caches and Local Memories

Lluc Alvarez, *Member, IEEE,* Lluís Vilanova, Marc Gonzàlez, *Member, IEEE,*
Xavier Martorell, *Member, IEEE,* Nacho Navarro, *Member, IEEE,* Eduard Ayguadé

Abstract—Cache coherence protocols limit the scalability of multicore and manycore architectures and are responsible for an important amount of the power consumed in the chip. A good way to alleviate these problems is to introduce a local memory alongside the cache hierarchy, forming a hybrid memory system. Local memories are more power-efficient than caches and do not generate coherence traffic, but they suffer from poor programmability. When non-predictable memory access patterns are found compilers do not succeed in generating code because of the incoherence between the two storages. This paper proposes a coherence protocol for hybrid memory systems that allows the compiler to generate code even in the presence of memory aliasing problems. Coherence is ensured by a software/hardware co-design where the compiler identifies potentially incoherent memory accesses and the hardware diverts them to the correct copy of the data. The coherence protocol introduces overheads of 0.26% in execution time and of 2.03% in energy consumption to enable the usage of the hybrid memory system, which outperforms cache-based systems by an speedup of 38% and an energy reduction of 27%.

Index Terms—Coherence protocol, Local memories, Scratchpad memories, Hybrid memory system

1 Introduction

Upcoming multicore and manycore architectures are expected to include a significant number of cores, as a result of the replication of general purpose and accelerator cores. As an immediate consequence, the memory subsystem has to evolve into some novel organization that overcomes the problems of traditional cache-based schemes. Two of the major concerns are the important amount of power consumed in the cache hierarchy and the lack of scalability of current cache coherence protocols, which constrain the sharing and the size of caches when cores are replicated beyond certain levels [1], [2], [3].

A possible solution to the power consumption and scalability problems of cache coherence protocols is the introduction of local memories (LMs), also known as scratchpad memories [4]. The main advantages of LMs are that they offer access delays similar to that of best-case cache delays in a much more power-efficient way and they do not generate coherence traffic. The drawback is that LMs introduce programmability difficulties due to the explicit data transfers they require, so usually programmers rely on compiler transformations that generate code to manage the LM. Although this limitation, LMs have been successfully introduced in the high performance computing (HPC) domain in several ways. In the Cell B.E. [5], accelerator cores access their private LM with memory instructions and use explicit DMA transfers to move data between memories. A more recent trend is to introduce a LM alongside the cache hierarchy, forming a hybrid memory system. This approach is currently used in GPGPUs [6].

One of the main problems of the hybrid memory system is the potential replication of data between the two storages. Compilers succeed in generating code for LMs when the computation is based on predictable memory access patterns [7] but, when non-predictable memory access patterns are found, compilers need to ensure correctness by applying complex analyses such as memory aliasing [8], [9], [10]. When compilers cannot ensure that there is no aliasing between two memory references that may target copies of the same data in the LM and in the cache hierarchy, they must conservatively avoid using the LM. This problem happens because the copies of data in the LM and in the cache hierarchy are incoherent.

The main contribution of this paper is a novel coherence protocol for hybrid memory systems to achieve the programmability of a cache-based system by safely enabling the use of the LM in the presence of memory aliasing problems. A coherent memory view of the two storages is ensured by a simple hardware/software mechanism implemented by two components: (1) a per-core hardware directory that keeps track of which data is mapped to the LM and (2) guarded instructions for memory operations that the compiler selectively places in potentially incoherent data accesses. At execution time the guarded memory instructions access the directory to identify which memory keeps the correct copy of the data and are diverted to it. The proposal allows the compiler to use an straightforward algorithm to generate code for the hybrid memory system. The evaluation shows that, compared to a compiler that is able to resolve all memory aliasing problems, the proposal introduces average overheads of 0.26% in execution time and 2.03% in energy consumption. These overheads are outweighted by the benefits coming from the ability to generate code for the hybrid memory system, that provides an average speedup of 38% and an average energy saving of 27% when compared to a cache-based system.

The rest of this paper is organized as follows: Section 2

• *L. Alvarez, L. Vilanova, M. Gonzàlez, X. Martorell, N. Navarro and E. Ayguadé are with the Department of Computer Architecture, Universitat Politècnica de Catalunya and the Barcelona Supercomputing Center, Barcelona, 08034 Spain. E-mail: name.surname@bsc.es*

gives some background of how a LM is integrated in a core and how the resulting architecture is programmed. Section 3 explains the design of the coherence protocol and Section 4 presents its evaluation. Section 5 comments some related work and Section 6 remarks the main conclusions of this work.

2 Background and Motivation

This section explains the hybrid memory system, its execution model and the coherence problem it exposes.

2.1 Baseline Architecture

The hybrid memory system consists of extending a core with a LM and a DMA controller (DMAC), as Figure 1 shows.

The LM is integrated into the core at the same level as the L1 cache and is used to store private data only. A range of the virtual address space is devoted to the LM, and this range is direct-mapped to the physical address space of the LM. The CPU needs three registers to keep track of the address mapping of the LM: a register for the base address of the virtual address range, a register for the base address of its physical address range and a register for the size of the LM. The CPU is able to access the LM using regular loads and stores to its virtual address range. In order to distinguish which memory has to serve a memory instruction, a range check is performed on the virtual address, prior to any MMU [11] action. If the virtual address is in the range reserved for the LM, the MMU is bypassed and a physical address that points to the LM is generated. This scheme is the preferred one to integrate a LM alongside the cache hierarchy [12], [13] because it has two important benefits. First, since no pagination is used for the LM, memory accesses to the LM do not need to access the TLB, so they are extremely power efficient and they have a deterministic latency. Second, it allows the introduction of the LM in a very simple way because only three registers are required to configure the LM and there is no interference with the cache hierarchy. In addition, the typical size of a LM is extremely small compared to the size of the RAM and of the virtual address space of a 64-bit machine, so the virtual and physical address ranges reserved for the LM occupy a very minor portion of the whole address spaces.

The DMAC is in charge of transferring data between the LM and the system memory (SM, which includes caches and main memory). It offers three operations: (1) *dma-get* transfers data from the SM to the LM, (2) *dma-put* transfers data from the LM to the SM and (3) *dma-synch* waits for the completion of certain DMA transfers. These operations are explicitly triggered by software using memory instructions to non-cacheable memory-mapped I/O registers in the DMAC. DMA transfers are coherent with the SM [14], [15] by inspecting the cache hierarchy at every bus request. The bus requests generated by a *dma-get* look for the data in the caches. If the data is in some cache, it is copied from there to the LM, otherwise it is copied from the main memory. The bus requests generated by a *dma-put* copy the data from the LM to the main memory and invalidate the cache line in the whole cache hierarchy, if it exists.

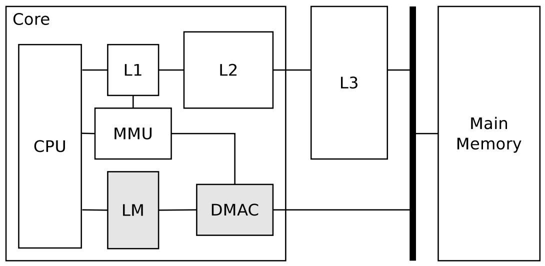


Fig. 1: Overview of the hybrid memory system. The architecture consists on extending a regular core with a Local Memory (LM) and a programmable DMA controller (DMAC).

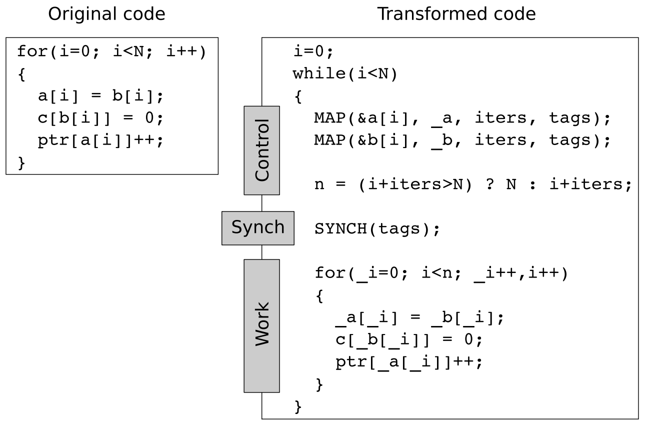
2.2 Execution Model

One of the big challenges of the hybrid memory system is to be more efficient than a cache-based system offering exactly the same level of programmability. Since the introduction of a LM imposes the software to explicitly manage the data, the only way to offer the programmer a system that is as programmable as a cache-based system is to give the compiler the responsibility of generating the code that manages the LM. In order to do so, the idea is that the programmer writes conventional parallel code and the compiler, first, identifies what data is better suited to be mapped to the LM and, then, generates code to manage this mapping transparently.

The first thing to be done by the compiler is to identify which data is private to each core, so it can be mapped to its LM. Typically programming models rely on the programmer to know how the data of a parallel program is distributed. In distributed memory architectures, programming models such as MPI [16] require the user to explicitly partition the data. Allocations are private to each computational task and the programmer adds explicit data transfers and synchronization points between tasks when needed. In shared memory architectures the programmer guides the partitioning. In OpenMP [17] the programmer adds code annotations to specify if the data is private or shared between threads and how the iteration space of a loop is split between the threads. Thus, in both cases, the data distribution between computational entities is solved by the inherent properties of the programming models themselves.

The data assigned to each core is then mapped to its LM, inducing a particular execution model. In the case of a computational loop, the code is converted into a two-level nested iterative structure that uses blocking [7], as Figure 2 shows. Each outermost iteration maps chunks of data to the LM and computes a subset of iterations. It executes three phases to do so: (1) a control phase that moves chunks of data between the LM and the SM, (2) a synchronization phase that waits for the DMA transfers to finish and (3) a work phase where the computation for the current chunk of data is performed. The three phases repeat until the whole iteration space is computed. These code transformations are usually done by run-time libraries [7], [18] or compilers [19], [20].

Automatic code transformations decide which data is mapped to the LM by analyzing the memory accesses [21]. *Regular accesses* are those that expose predictable access patterns (e.g., with a constant stride). These are mapped to



```
for(i=0; i<N; i++)
{
  a[i] = b[i];
  c[b[i]] = 0;
  ptr[a[i]]++;
}
```

```
i=0;
while(i<N)
{
  MAP(&a[i], _a, iters, tags);
  MAP(&b[i], _b, iters, tags);

  n = (i+iters>N) ? N : i+iters;

  SYNCH(tags);

  for(_i=0; i<n; _i++,i++)
  {
    _a[_i] = _b[_i];
    c[_b[_i]] = 0;
    ptr[_a[_i]]++;
  }
}
```

Fig. 2: Code transformation for the hybrid memory system and three-phase execution model of the transformed code.

the LM. Unpredictable memory accesses are difficult to map to the LM [7], so they are served by the cache hierarchy. These are called *irregular accesses*. In the original code in Figure 2, the accesses to `a` and `b` are regular accesses, and the accesses to `c` and `ptr` are irregular.

In the control phase, chunks of data are moved between the LM and the SM. In order to do this task in a simple and efficient way, the compiler declares as many buffers in the LM as regular accesses appear in the loop. All buffers have the same size, determined by the size of the LM and the number of regular accesses. In Figure 2 there are two regular accesses (`a` and `b`) so two buffers (`_a` and `_b`) would be allocated in the LM, each one of them occupying half the storage. In every instance of the control phase, for each regular access, the chunk of data that is needed in next work phase is mapped to its corresponding LM buffer (`MAP` statements in Figure 2), potentially sending back to the SM the previously used chunk. Even in case of mapping a chunk of data to the LM for writing only, the transfer of the chunk from the SM to the LM is done because otherwise, if only part of the chunk was modified, the write-back to the SM would update the unmodified parts of the copy in the SM with garbage.

The work phase is like the original loop, but with two differences. First, every instance of the work phase consumes a subset of the original iteration space. The amount of iterations depends on the stride of the regular accesses and the size of the LM buffers. Second, the original regular accesses (`a` and `b`) are substituted with their LM buffer counterparts (`_a` and `_b`) while irregular accesses are left untouched (`c` and `ptr`).

2.3 The Coherence Problem

The coherence problem in the hybrid memory system appears when two incoherent copies of the same data can be accessed during the computation. When some data is mapped to the LM, a copy of the data is created. For regular accesses, the compiler generates memory operations that access the copy in the LM while, for irregular accesses, it generates memory operations that access the copy in the SM. Since the memories are incoherent, modifications are not visible between paths, so the execution can be incorrect.

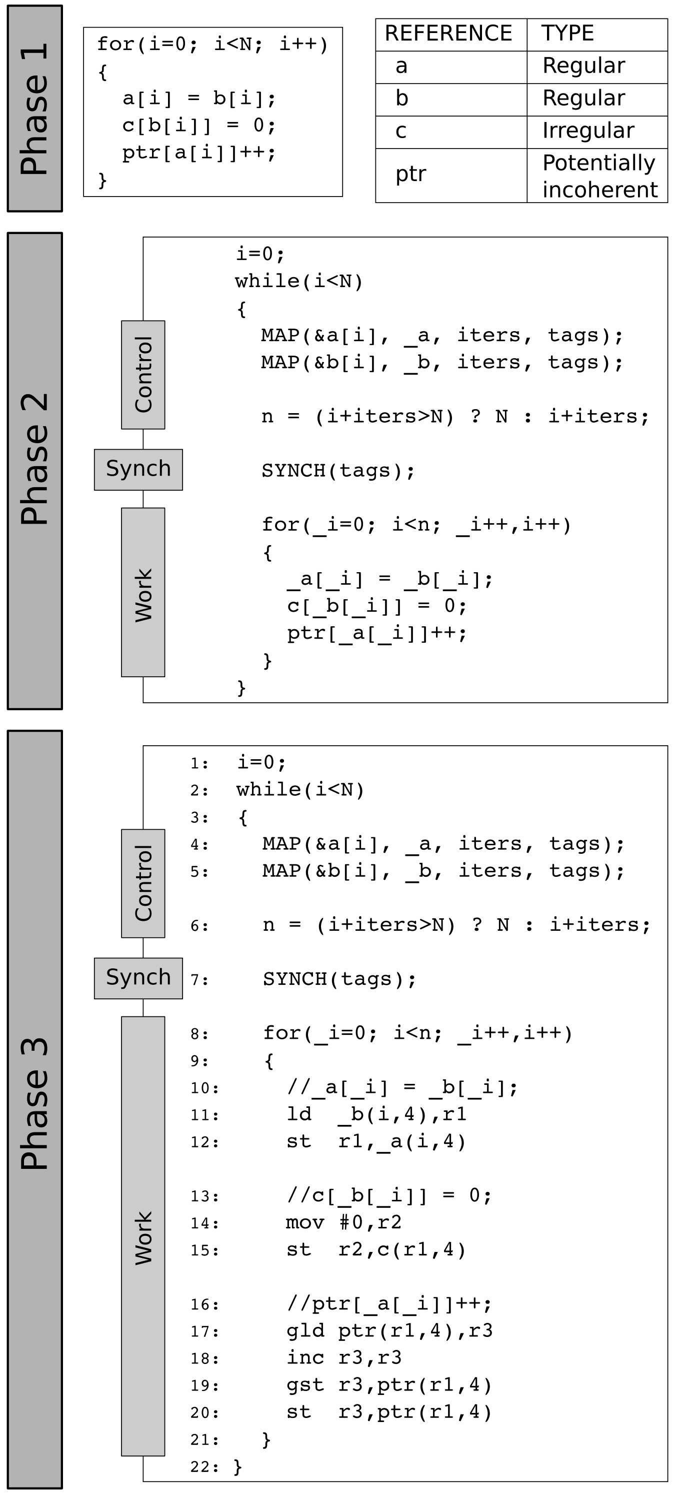
Compiler-based solutions for this situation are inefficient. All approaches rely on memory aliasing analyses [8], [9], [10]. In Figure 2 this means predicting when, if ever, any instance of the accesses to `c` or `ptr` aliases with any instance of the accesses to `a` or `b`. Current algorithms are not able to solve this problem in the general case, so compilers adopt restrictive solutions in its presence. The naive one is to discard the usage of the LM in presence of a *potentially incoherent access*. A potentially incoherent access is an irregular access that the compiler cannot ensure it will never access data in the SM that is mapped to the LM. Another option is to introduce fine-grained DMA transfers surrounding the potentially incoherent accesses [7], adding big overheads because DMA transfers of small sizes are inefficient. Software caching is another solution [22], [7]. These keep track of the contents of the LM with a software directory and perform a costly associative search on it prior to every potentially incoherent access to determine if the access has to go to the LM or to the SM.

This paper proposes an efficient mechanism that ensures coherence in hybrid memory systems. The solution avoids the limitations stemming from the inability to solve the memory aliasing problem, bringing the optimization opportunities to a new level where automated optimization tools no longer have to back-off their code transformations due to coherence issues.

3 DESIGN

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 LM and the cache hierarchy. The system guarantees that, first, in case of data replication, either the copies are identical or the copy in the LM 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, potentially incoherent accesses are diverted to the memory that keeps the valid copy. In order to do so a directory is introduced to keep track of what data is mapped to the LM. The DMAC updates the directory entries when it executes *dma-get* commands. The compiler identifies potentially incoherent memory accesses and emits *guarded memory instructions* for them. The execution of a guarded memory instruction triggers a lookup in the directory, diverting the access to the memory that keeps the valid copy of the data.

The proposed coherence protocol is independent of the cache coherence protocol. The proposed coherence protocol is per core and it ensures coherence between the caches and the LM of that core, without interacting with other cores nor with the cache coherence protocol. The proposed coherence protocol can be integrated in a multicore with the hybrid memory system by simply replicating the per core hardware support in every core. This is because the LMs in the hybrid memory system are used to store per core private data only. One core cannot access the LM of another core and, when a core maps data to its LM, another core should not access



```
for(i=0; i<N; i++)
{
  a[i] = b[i];
  c[b[i]] = 0;
  ptr[a[i]]++;
}
```

REFERENCE	TYPE
a	Regular
b	Regular
c	Irregular
ptr	Potentially incoherent

```
i=0;
while(i<N)
{
  MAP(&a[i], _a, iters, tags);
  MAP(&b[i], _b, iters, tags);

  n = (i+iters>N) ? N : i+iters;

  SYNCH(tags);

  for(_i=0; i<n; _i++,i++)
  {
    _a[_i] = _b[_i];
    c[_b[_i]] = 0;
    ptr[_a[_i]]++;
  }
}
```

```
1:  i=0;
2:  while(i<N)
3:  {
4:    MAP(&a[i], _a, iters, tags);
5:    MAP(&b[i], _b, iters, tags);

6:    n = (i+iters>N) ? N : i+iters;

7:    SYNCH(tags);

8:    for(_i=0; i<n; _i++,i++)
9:    {
10:     //_a[_i] = _b[_i];
11:     ld  _b(i,4),r1
12:     st  r1,_a(i,4)

13:     //c[_b[_i]] = 0;
14:     mov #0,r2
15:     st  r2,c(r1,4)

16:     //ptr[_a[_i]]++;
17:     gld ptr(r1,4),r3
18:     inc r3,r3
19:     gst r3,ptr(r1,4)
20:     st  r3,ptr(r1,4)
21:   }
22: }
```

Fig. 3: Example of application of the three phases of the compiler support. In the first phase the memory references are classified as regular, irregular or potentially incoherent. In the second phase the code is transformed to follow the execution model for the hybrid memory system. In the third phase the assembly code is emitted, generating guarded memory instructions for the potentially incoherent memory instructions (lines 17 and 19) and the double store if needed (line 20).

the copy of this data in the SM. This is key to ensure there is no interaction with the cache coherence protocol and it is what allows the proposal to work by only monitoring events inside the core. This constraint is easily ensured when a compiler maps private data to the LM because the data distribution is already specified in the parallelization model. If the architecture is programmed by hand, the programmer is responsible for not accessing the data mapped to one core from another core without using synchronization primitives.

The next sections explain the compiler and hardware support for the coherence protocol, show an example of how everything works together and describe of how the system manages the copies of the data, in such a way that the conditions for the correctness of the coherence protocol are always fulfilled.

3.1 Compiler Support

With the proposed coherence protocol the compiler algorithm that transforms the code as shown in Figure 2 is straightforward and safe, even in the presence of memory aliasing problems. The compiler support, as shown in Figure 3, consists on three phases: classification of memory references, code transformation and code generation.

Phase 1 - Classification of memory references: In this phase the compiler identifies which memory accesses are suitable to be mapped to the LM and which others to the SM. 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:

- *Regular accesses* are those that expose a strided access pattern. They access the LM.
- *Irregular accesses* are those that do not expose a strided access pattern and the compiler determines they do not alias with any regular access. They access the SM.
- *Potentially incoherent accesses* are those that do not expose a strided access pattern and the compiler determines they alias or may alias with some regular access. They access the directory and then the SM or the LM.

In the example shown in Figure 3, the compiler classifies `a` and `b` as regular 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 irregular or as potentially incoherent accesses. The example assumes the compiler succeeds in ensuring that `c` does not alias with any regular access and that it is unable to do so for `ptr`, so it classifies `c` as an irregular access and `ptr` as a potentially incoherent access.

Phase 2 - Code transformation: In this phase the compiler transforms the code for regular accesses as explained in Section 2.2. These are typical transformations to manage LMs using tiling [20], [7]. For irregular and potentially incoherent accesses nothing is done in this phase.

In Figure 3 the code is transformed in exactly the same way as explained in Section 2.2. The only difference is the existence of a new class of memory accesses, the potentially incoherent ones, like `ptr`. Since the compiler does not have to do any transformations for them, the resulting code is the same as in Figure 2.

Phase 3 - Code generation: In this phase the compiler generates the assembly code for the target architecture:

- For *regular accesses* the compiler generates memory instructions that directly access the LM. This is accomplished by using as source operands the base address of a LM buffer and an offset.
- For *irregular accesses* the compiler generates memory instructions that directly access the SM. This is accomplished by using as source operands a base address in the SM and an offset.
- For *potentially incoherent accesses* the compiler generates guarded memory instructions with an initial SM address. This is accomplished by using as source operands a base address in the SM and an offset. When it is executed, the guarded memory instruction accesses the directory using the SM address and is diverted to the corresponding memory. The implementation of the guarded memory instruction is discussed later in this section.

Figure 3 shows the assembly code that the third phase emits for the body of the innermost loop. In the statement that uses regular 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, its addresses will guide the memory accesses to the LM. Similarly, in the statement that uses an irregular 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 SM 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 SM addresses based on `ptr` will be used to look up the directory and they will be changed to LM addresses if a copy of the data exists there.

One special case has to be treated separately. When the compiler determines a write access is potentially incoherent it has to ensure also that the access does not alias with some data that is mapped to the LM as read-only. If it cannot ensure this, emitting a single guarded store can lead to an erroneous execution. This is caused by a typical optimization of tiling transformations that consists on not triggering a write-back to the SM of a chunk of data that is mapped as read-only. With this optimization, what will happen at execution time is that the guarded store will hit in the directory and the modification will be done to the LM. Since the buffer will not be written-back to the SM, when the buffer is reused to map new data, the corresponding *dma-get* operation will overwrite the contents of the buffer and the modifications done by the potentially incoherent store will be lost. A naive solution to this problem is to disable the tiling optimization, forcing that the write-back is always performed and so incurring in high performance penalties. A more efficient solution is to make the modifications in the two memories. A simple way to do it is that the compiler generates a *double store*: one irregular store that will update the copy in the SM and one potentially incoherent store that will trigger a lookup in the directory and will update the copy in the LM if it exists. Note that if the lookup in the directory of the potentially incoherent store misses there will be two stores of the same data to the same SM address. The overhead of this unnecessary second store is small. The performance impact is low because the two stores are independent so they both can be issued in the same cycle. The increase in power consumption is also small since the Load/Store Queue [11] will collapse the second store with the first one if it is not yet committed, having one single cache access and so not paying the cost of an extra memory access. Note also that, in presence of a potentially incoherent store, the compiler almost always generates a double store since, in general, it is unable to ensure that the aliasing is not with some read-only data. This happens because typically the compiler is unable to determine what is the accessible address range of a potentially incoherent access, therefore it is also unable to ensure there is no read-only data in this potentially infinite accessible address range.

The final code of Figure 3 shows how the compiler generates the double store. For the increment of a random position of `ptr` (line 16), the value is read with a guarded load (`gld` in line 17), incremented and finally written with a double store. The double store consists of a guarded store (`gst` in line 19) that will modify the copy in the LM if it exists and a conventional store (`st` in line 20) with the same source operands that will always update the value in the SM.

The implementation of the guarded memory operations is highly architecture-dependent. 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 [23]. Another option is to provide a fewer range of guarded memory instructions and restrict the compiler to these. In CISC architectures like x86 [24], 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 directory 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.

3.2 Hardware Design

The only hardware support needed for the coherence protocol is a directory that keeps track of the contents of the LM. This section explains how the directory is configured, updated and used in the address generation. Then some considerations about its access time, its double buffering support and its side effects on the hybrid memory system are discussed.

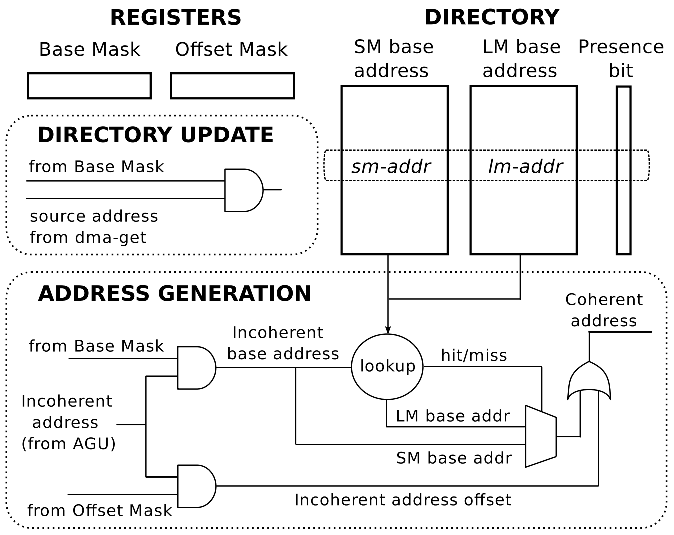


Fig. 4: Scheme and operations of the directory for the coherence protocol. The directory is updated at every *dma-get*. In the address generation stage of the guarded memory instructions, the directory is looked up to generate a SM or a LM address.

Configuration: The directory can be configured to work with any LM buffer size. When the compiler transforms the code it partitions the LM into equally sized buffers and informs the hardware of the LM buffer size through a memory-mapped register. A directory entry is assigned to each of these LM buffers to map the starting address of the copy of the data in the SM (i.e., the directory tag) to the starting address of the LM buffer where the data is mapped to. Since all LM buffers are equally sized, the base address of a LM buffer is equivalent to the buffer number and, thus, the index of a directory entry. The buffer size is used to set the values of the *Base Mask* and *Offset Mask* internal registers. These registers allow to decompose any address into a base address and an address offset, so the directory can be operated with any buffer size.

Update: Every *dma-get* operation updates the directory. The destination LM address of the transfer is used to identify the base address of the LM buffer and the source SM address is used to set the tag of the corresponding directory entry.

Address generation: The directory is used in the address generation as shown in Figure 4. The Address Generation Unit (AGU) [11] first generates a potentially incoherent SM address (*Incoherent address*). Notice that this is a SM address because it is generated by a potentially incoherent access. Two bit-wise AND operations between the *Incoherent address* and the *Base Mask* and *Offset Mask* registers split the address in an *Incoherent base address* and an *Incoherent address offset*. The *Incoherent base address* is used to do a lookup in the directory. If it hits, the instruction is accessing data in the SM that has a copy in the LM, so the access has to be diverted to the LM. The base address of the corresponding LM buffer is retrieved from the directory (*LM base addr*) and a bit-wise OR with the *Incoherent address offset* is done, resulting in the *Coherent address*. If the lookup misses there is no copy in the LM, so the original SM address is preserved performing a bit-wise OR between the *SM base addr* and the *Incoherent address offset*.

Access time: The directory is restricted to have 32 entries to keep the access time low. According to CACTI [25], with a process technology of 45nm, the latency of the directory is 0.348 ns. Taking into account that this latency would be significantly lower with nowadays process technology, that current CPUs work with frequencies between 2GHz and 3GHz and that the directory is accessed just after an extremely simple operation in the AGU, it is feasible to generate the address and to do the lookup in the same cycle. Having 32 entries constrains the software to use 32 LM buffers at most, so loops can only map 32 regular references to the LM. This is not a big limitation since loops with more than 32 regular references are rare. If a loop needs more than 32 buffers the compiler can simply not map the exceeding regular accesses to the LM.

Double buffer support: The directory contains a *Presence bit* that indicates if the data of a LM buffer is currently being transferred into the LM by a *dma-get*. This bit is reset when the *dma-get* is triggered. If a guarded memory access hits the directory entry and this bit is unset, an internal exception is generated until the bit is set at the *dma-get* completion. This ensures correctness when a guarded memory access accesses data that is being transferred to the LM using double buffering.

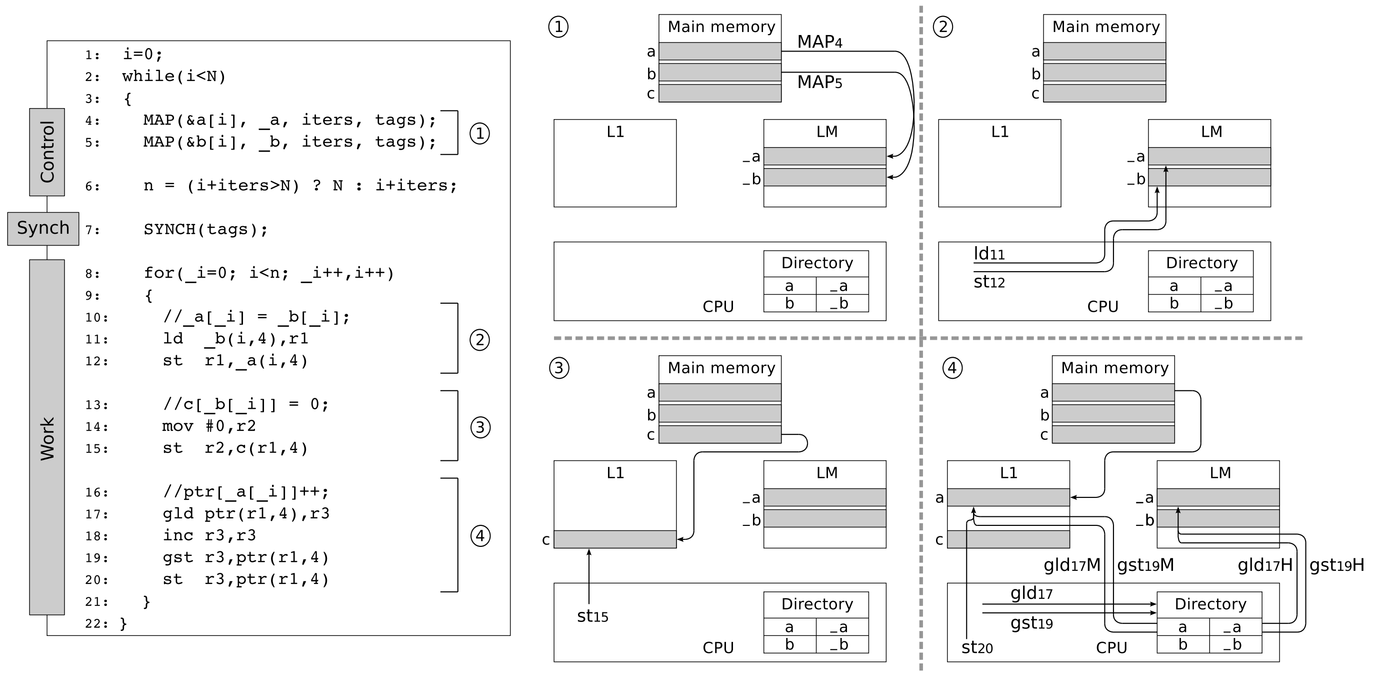
As a final remark, the introduction of the hardware directory does not undermine the benefits of the hybrid memory system. The number of CAM lookups is kept low because only accesses that are not regular trigger them: if they are potentially incoherent accesses they go through the directory and then to either the cache or the LM; if they are irregular accesses they are served directly by the cache. Regular accesses are directly served by the LM without any CAM lookup. Since in HPC applications the vast majority of memory accesses are regular [26], [27], the directory is rarely accessed and the goodnesses of the hybrid memory system are preserved.

3.3 Example of Operation

The cooperation between the compiler support and the hardware additions for the coherence protocol achieve that the memory accesses are always served by a memory that keeps a valid copy of the data. This section shows an example of how the whole mechanism operates together in order to do so.

Figure 5 shows an example of operation. The leftmost part of the figure shows the final code generated by the compiler after applying the three-phase transformations explained in Section 3.1. This code is the same as the resulting code of Figure 3. The rightmost part of the figure shows how the hardware executes the code. The execution is divided in four steps that correspond to four pieces of code. For every step the figure explains which memory serves the memory operations triggered by the corresponding piece of code. Note that the memory operations are labeled indicating the instruction and the line of the code that triggers them (e.g., `MAP4` represents the memory operation triggered by the statement `MAP` of the 4th line of code). Note also that, for simplicity, the cache hierarchy in the figure only shows the first level of cache.

The execution starts with the step 1. Its first statement (line 4) maps a chunk of `a` to the LM buffer `_a`. At execution time, the transition `MAP4` shows how a DMA transfer makes a copy


```
 1:  i=0;
 2:  while(i<N)
 3:  {
 4:    MAP(&a[i], _a, iters, tags);
 5:    MAP(&b[i], _b, iters, tags);

 6:    n = (i+iters>N) ? N : i+iters;

 7:    SYNCH(tags);

 8:    for(_i=0; i<n; _i++,i++)
 9:    {
10:      //_a[_i] = _b[_i];
11:      ld  _b(i,4),r1
12:      st  r1,_a(i,4)

13:      //c[_b[_i]] = 0;
14:      mov #0,r2
15:      st  r2,c(r1,4)

16:      //ptr[_a[_i]]++;
17:      gld ptr(r1,4),r3
18:      inc r3,r3
19:      gst r3,ptr(r1,4)
20:      st  r3,ptr(r1,4)
21:    }
22: }
```

Fig. 5: Example of the hardware handling memory operations in the hybrid memory system with the coherence protocol. The code is divided in four pieces, each one of them corresponding to a step in the execution diagram. Every step shows how the data is moved between memories and what memory serves the memory accesses triggered by the corresponding piece of code.

from `a` in the SM to `_a` in the LM. It is assumed the caches are empty, so the data is transferred from the main memory. If some cache kept the valid copy of the data, the coherent DMA transfers would read the data from the cache. Similarly, the second statement of this step (line 5) maps a chunk of `b` to `_b`, which at execution time provokes the data movement from the main memory to the LM represented by `MAP5`.

Once the control phase has been executed, the step 2 takes place. This step is the assignment from `_b` to `_a` done with regular accesses (line 10). In assembly language the assignment is done using a conventional load (line 11) and a conventional store (line 12). At execution time, the load to `_b` is served directly by the LM, as represented by `ld11`, and the store to `_a` is also directly sent to the LM, as `st12` represents. These direct accesses to the LM happen because the addresses of the memory operations are in the range reserved to the LM.

In the step 3 an irregular store sets to zero a random position of `c` (line 13). The store is irregular because it does not expose a strided access pattern and the example assumes the compiler can ensure the access does not alias with any regular access. The assembly code for this statement consists on placing the value zero in a register (line 14) and then storing this value to memory using a conventional store to some position of `c` (line 15). This store, labeled as `st15`, is served by the L1 cache at execution time, since the address it modifies is not in the LM address space. Assuming the caches are empty, the L1 cache requests the cache line to the upper levels of the hierarchy (not shown in the figure) and these forward the request to the main memory. The cache line is then sent to the requesters, reaching the L1 cache so the modification can be done.

Finally, the step 4 increments an element of `ptr` using potentially incoherent loads and stores (line 16). These memory accesses are potentially incoherent because they are not strided and the example assumes the compiler does not succeed in ensuring they do not alias with any regular access. In addition, the potentially incoherent write access needs to be treated with a double store. The instructions for this statement are a guarded load of some element of `ptr` (line 17), the increment (line 18), a guarded store that will write the new value in the LM in case it exists (line 19) and a conventional store that will always write the new value in the SM (line 20). When these instructions are executed, the guarded load `gld17` does a lookup in the directory. If it hits, the access is diverted to the LM as `gld17H` shows. This happens, for instance, if `ptr` equals `a` and `ptr[_a[_i]]` is a position of `a` that has been mapped to the LM in the step 1. Otherwise, if the directory lookup misses, the load labeled as `gld17M` is served by the L1 cache, which requests the cache line if needed. This happens, for instance, if `ptr` equals `a` but `ptr[_a[_i]]` is a position of `a` that has not been mapped to the LM in the step 1. After loading the value, it is incremented and written to memory with the guarded store `gst19`. The execution of the guarded store, analogous to the guarded load, first does a lookup in the directory. If there is a copy of the data in the LM the lookup hits, the address is changed to point to the LM and the access goes there as `gst19H` shows. If there is no copy in the LM the lookup misses and the SM address is preserved, so the L1 cache serves the access as `gst19M` shows. With this mechanism, always the valid copy of the data is accessed. To prevent losing the modifications when there is aliasing with read-only data in the LM, the irregular store `st20` modifies the copy in the SM. The address of the store guides the operation to the L1 cache, which requests the cache line if necessary.

3.4 Data Coherence Management

This section shows the correctness of the coherence protocol. The two previous sections described how memory operations are diverted to one memory or another when replication exists, considering that the valid copy of the data is in the LM. 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 the LM and in the cache hierarchy, only two situations can arise: either both versions are identical, or the version in the LM 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 LM 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.

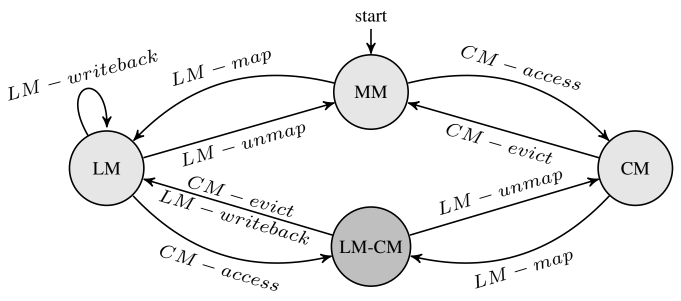


Fig. 6: State diagram of the possible replication states of the data. A piece of data can be in main memory only (MM), replicated only in the LM (LM), replicated only in the cache hierarchy (CM), or replicated in the LM and in the cache hierarchy at the same time (LM-CM). Creating and discarding copies of the piece of data cause transitions between the states.

3.4.1 Data States and Operations

Figure 6 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 the LM. The *LM* state indicates that only one replica exists, and it is located in the LM. In the *CM* state only one replica in the cache hierarchy exists. In the *LM-CM* state two replicas exist, one in the LM and the other in the cache hierarchy.

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

The *MM*→*LM* transition occurs when the software causes an *LM-map* action. Switching back to the *MM* state occurs when an *LM-unmap* action happens due to a *dma-get* mapping new data to the buffer. Notice that an *LM-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 LM.

Transitions between the *MM* and *CM* states happen according to the execution of load and store operations that cause *CM-access* and *CM-eviction* actions. Notice that unless the data reaches the *LM-CM* state, no coherence problem can appear due to the use of a LM. DMA transfers are coherent with the SM, ensuring the system coherence as long as the data switches between the *LM* 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 *LM-CM* state is reachable from both the *LM* and the *CM* states. In the *LM* state, a guarded instruction will never cause a replica in the caches since the access goes through the directory, and this will divert the access to the LM. It is impossible to have unguarded memory instructions to the SM because the compiler never emits them unless it is sure that there is no aliasing, which cannot happen in this state. In the *LM* state, only the execution of a double store can cause the transition to the *LM-CM* state. The double store is composed of a guarded store and a store to the SM ($\texttt{st}_{\texttt{guarded}}$ and $\texttt{st}_{\texttt{sm}}$). The $\texttt{st}_{\texttt{sm}}$ is served by the cache hierarchy, so a replica of the data is generated and updated in the cache, while the $\texttt{st}_{\texttt{guarded}}$ modifies the LM replica with the same value, so two replicas generated through a *LM*→*LM-CM* transition are always identical. The transition *CM*→*LM-CM* happens due to an *LM-map* action, and the DMA coherence ensures the two versions are identical. Once in the *LM-CM* state, the double store updates both versions, while $\texttt{st}_{\texttt{guarded}}$ and $\texttt{st}_{\texttt{lm}}$ modify the LM version and $\texttt{st}_{\texttt{sm}}$ will never be generated.

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

3.4.2 Data Eviction

The state diagram shows that the eviction of data can only occur from the *LM* and *CM* states. There is no direct transition from the *LM-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 *LM-CM* state, its eviction can only occur if first one of the replicas is discarded, which corresponds to a transition to the *LM* or *CM* states. According to the previous section, it is ensured that in the *LM-CM* state the two replicas are identical or, if not, the version in the LM 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

TABLE 1: PTLsim configuration parameters.

Parameter	Description
Pipeline	Out-of-order, 4 instructions wide
Branch predictor	Hybrid 4K selector, 4K G-share, 4K Bimodal 4K BTB 4-way, RAS 32 entries
Functional units	3 INT ALUs, 3 FP ALUs, 2 load/store units
Register file	256 INT registers, 256 FP registers
L1 I-cache	32 KB, 8-way set-associative 2 cycles latency
L1 D-cache	32 KB, 8-way set-associative write-through, 2 cycles latency
L2 cache	256 KB, 24-way set-associative write-back, 15 cycles latency
L3 cache	4 MB, 32-way set-associative write-back, 40 cycles latency
Prefetcher	IP-based stream prefetcher [30], [31] to L1, L2 and L3
Local memory	32 KB, 2 cycles latency

the *LM-CM* state. When a *LM-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 SM that causes a replacement of the cache line that holds the current data, leaving just one replica, the one in the LM, and thus transitioning to the *LM* state. Once the *LM* state is reached, at some point the program will execute a *dma-put* operation to write-back the data to the SM. Finally, the transition *LM-CM*$\rightarrow$*CM* caused by a *LM-unmap* action corresponds to the case where the program explicitly discards the copy in the LM when new data is mapped to the buffer that holds it. The programming model imposes that this will only happen when both versions are identical, because if the version in the LM had modifications it would be written-back before being replaced. So, after the *LM-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 line 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 Evaluation

This section evaluates the coherence protocol for the hybrid memory system. A microbenchmark and a set of real benchmarks are used to study the overhead of the proposal in terms of execution time and energy consumption. Then a comparison against a cache-based system is presented.

4.1 Experimental Framework

The proposal has been evaluated using PTLsim [28], extending it with a LM, a DMAC and the directory of the coherence protocol. For the energy results Wattch [29] has been integrated into the simulator. Single-core simulations are presented because the coherence protocol is per core. Table 1 shows the parameters of the simulated speculative out-of-order core.

Six memory intensive HPC benchmarks from the NAS benchmark suite [32] are used for the evaluation. The benchmarks have been compiled using GCC 4.6.3 with the -O3 optimization flag on. SimPoint [33] has been used to identify the simulation points and at least 150 millions of x86 instructions have been simulated for each benchmark.

The outcome of the alias analysis performed by GCC on every memory reference has been checked to generate the guarded memory instructions. The references that GCC is not able to determine the aliasing for are the potentially incoherent accesses. Once these accesses have been identified, the source code of the benchmarks has been modified by hand to generate the guarded memory instructions using assembly macros. x86 instruction prefixes are used to implement the guarded instructions as explained in Section 3.1.

TABLE 2: Scheme of the microbenchmark. The microbenchmark is a simple loop that can be configured in four modes. For each mode it is assumed some memory references are potentially incoherent so guarded memory instructions are emitted for them, represented with bold font.

Microbenchmark	Mode	Assembly code
	Baseline	`mov a(,esi,4),ebx`
		`add edi,ebx`
		`mov ebx,a+4(,esi,4)`
`int a[N];`	RD	**`mov a(,esi,4),ebx`**
`int c;`		`add edi,ebx`
`for(i=0; i<N-1; i++)`		`mov ebx,a+4(,esi,4)`
`{`	WR	`mov a(,esi,4),ebx`
`  a[i+1] = a[i] + c;`		`add edi,ebx`
`}`		**`mov ebx,a+4(,esi,4)`**
		`mov ebx,a+4(,esi,4)`
	RD/WR	**`mov a(,esi,4),ebx`**
		`add edi,ebx`
		`mov ebx,a+4(,esi,4)`
		`mov ebx,a+4(,esi,4)`

4.2 Overhead of the Coherence Protocol

A microbenchmark that stresses the coherence protocol is used to facilitate the study of its performance overheads. Table 2 shows its characteristics. The microbenchmark is a loop that makes a sequence of load/add/store instructions that can be configured in four modes. In the baseline mode no guarded instructions are generated for any access. The RD mode assumes the read access `a[i]` is potentially incoherent, so a guarded load is generated. The guarded memory instructions are represented in bold font in the assembly code. The WR mode assumes the write access to `a[i+1]` is potentially incoherent and it cannot be ensured a write-back to the SM will be performed, so a double store is emitted. The RD/WR mode is a combination of the RD and the WR modes. To model all possible scenarios in terms of the ratio of accesses that are potentially incoherent, the percentage of memory operations that need to be guarded can also be adjusted.

Figure 7 shows the overhead in execution time of the proposal in the microbenchmark. Three lines appear in the figure, one per each mode of the microbenchmark. The X axis shows the percentage of references that are potentially incoherent with respect to the total number of references. The overhead of each mode is shown in ratio and computed against the baseline mode of the microbenchmark.

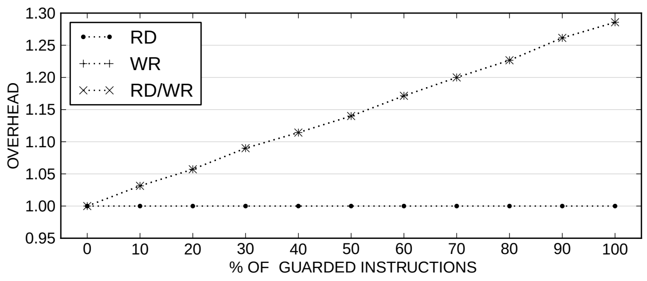


Fig. 7: Overhead in all microbenchmark modes.

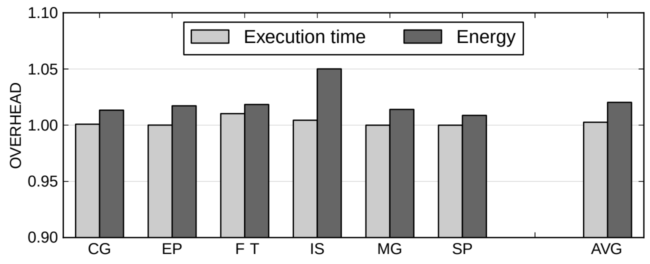


Fig. 8: Overhead in real benchmarks.

The RD mode line shows no overhead at all. The only differences in the execution of a guarded load and a non-guarded load are that the prefix has to be decoded and that a lookup in the directory is triggered. Both operations fit in the cycle time so there is no performance overhead for guarded loads. In the WR and the RD/WR modes it can be observed a linear overhead as the percentage of potentially incoherent accesses grows. The overhead is caused by the extra store added. When the double store is used at every write access it adds an overhead of 28%, which is provoked by an increase in executed instructions of 26%. The double store also adds pressure to the Load/Store Queue, although not enough to become a bottleneck. The overhead decreases to less than 10% when 35% or less of the write access are guarded and need the double store, which provokes an increase of 9% in executed instructions. Notice that in the WR and RD/WR modes, if the compiler could ensure the potentially incoherent write access aliases with some data in the LM that will be written back to the SM, a single guarded store would be generated instead of the double store, and the overhead would be zero as in the case of a single guarded load.

In conclusion, the coherence protocol adds no performance overhead when the potentially incoherent memory accesses are for reading data or when they are for writing and the double store is not needed. Only the double store adds overhead, reaching a maximum 28% in the microbenchmark. In real situations it is common that the number of potentially incoherent write accesses is low with respect to the total number of memory accesses and the computation is more complex than the one performed in the microbenchmark, so the expected overheads are far from this reported upper bound.

In order to study the overheads in real benchmarks, the hybrid memory system extended with the coherence protocol is compared against an incoherent hybrid memory system with an oracle compiler. In this baseline architecture the potentially incoherent accesses are left unguarded and are always served by the memory that has the valid copy of the data.

Figure 8 shows the overhead introduced by the coherence protocol in terms of execution time and energy consumption in real benchmarks. The performance overhead in CG, MG and SP is zero because the compiler does not find any potentially incoherent write access that needs to be treated with a double store. This happens only in FT and IS, which present overheads of 1.03% and 0.44%, respectively, and in EP, which presents no overhead. FT uses 34 strided references, 2 potentially incoherent read references and 2 potentially incoherent write references (treated with a double store) to do complex operations on floating point data. The cost of the computation and the small percentage of references that need to be treated with the double store keep the overhead low. In IS the computation is very simple and the double store is used in 2 out of 5 references, so the extra store provokes a non-negligible increase in the number of executed instructions. These extra instructions barely affect the performance because most of the times the out-of-order engine is able to issue the potentially incoherent store and the irregular store in the same cycle, effectively hiding the performance penalty caused by the double store. A similar situation happens in EP, that has 3 strided references, 16 local variables and 1 potentially incoherent write reference for which the double store is used. In this case the issue of the two stores is always done in the same cycle, that is why the overhead is zero. The resulting average overhead of the benchmarks is negligible, 0.26%.

Figure 8 also shows the energy consumption overhead is less than 2% in all benchmarks except in IS. These benchmarks have many strided references and do complex computations, so the directory is very seldomly accessed and, moreover, the energy it consumes is much lower than the energy consumed by other components such as the memory subsystem, ALUs and issue queues, resulting in a very low overhead. In IS the overhead is 5%. The overhead generated by the directory is around 1.8%, the remaining 3.2% is caused by the execution of the double store. The average overhead in energy consumption of all benchmarks is 2.03%.

In conclusion, the coherence protocol adds a very low overhead in performance and in energy consumption. In 3 of the 6 benchmarks the double store is not needed, so there are no performance penalties and the utilization of the directory generates an increase in energy consumption of less than 2%. When the double store is needed the increase in the number of instructions provokes a very minor performance degradation and a slightly higher energy consumption.

4.3 Comparison with Cache-Based Architectures

The immediate result of the coherence protocol is that any computational kernel can now be executed on the hybrid memory system no matter the restrictions coming from coherence problems. In order to show the usefulness of this achievement, this section evaluates the benefits in performance and energy consumption of the coherent hybrid memory system when compared to a cache-based system.

The coherent hybrid memory system and the cache-based system studied in this section have the same characteristics but

TABLE 3: Activity in the memory subsystem for the hybrid memory and the cache-based systems.

Benchmark		Guarded	AMAT	L1		L2	L3	LM	Directory
Name	Mode	References		Hit ratio	Accesses	Accesses	Accesses	Accesses	Accesses
CG	Hybrid coherent	1/7 (14%)	3.15	90.52	19319	26376	10597	30235	10566
CG	Cache-based	0	4.31	82.23	70371	62822	84202	0	0
EP	Hybrid coherent	1/20 (5%)	2.14	99.93	37152	10266	228	3862	3519
EP	Cache-based	0	2.37	98.93	43814	13219	797	0	0
FT	Hybrid coherent	4/34 (11%)	2.60	96.61	912779	761009	110186	1155150	55118
FT	Cache-based	0	4.95	78.54	1379688	789765	352269	0	0
IS	Hybrid coherent	2/5 (25%)	6.27	74.00	140663	194465	74647	73400	25714
IS	Cache-based	0	7.93	64.10	169425	182716	127692	0	0
MG	Hybrid coherent	1/60 (1.66%)	2.24	99.71	605269	252799	35588	798562	19377
MG	Cache-based	0	3.89	90.65	827239	238099	127176	0	0
SP	Hybrid coherent	0/497 (0%)	2.41	98.37	331832	162441	24159	235024	0
SP	Cache-based	0	4.73	79.59	407952	164515	82301	0	0

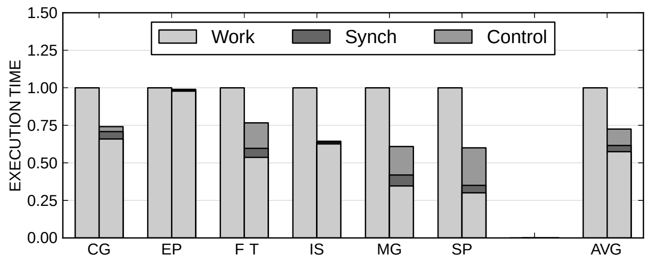


Fig. 9: Reduction in execution time.

with one difference. The hybrid memory system has a 32KB LM and the directory of the coherence protocol. For fairness, the capacity of the L1 of the cache-based system is increased to 64KB, matching the 32KB of LM plus the 32KB of L1 in the hybrid memory system. Table 3 summarizes the statistics of the memory subsystem that are the dominating factors of the improvements. This table is used throughout this section to explain the differences between the two architectures. For each benchmark the table shows the ratio of references that are potentially incoherent, the average memory access time (AMAT), the L1 hit ratio and the number of accesses to all the components of the memory subsystem in thousands. The accounting of accesses includes hits, misses, lookups and invalidations provoked by memory instructions, prefetchers, placement of cache lines by the MSHRs, write-through and write-back policies and bus requests of the DMA commands.

The immediate consequence of the coherence protocol is that any computational loop can be executed on the hybrid memory system. The benchmarks that take benefit of this achievement are all but SP. In Table 3 this is reflected in the column of the number of guarded references. All benchmarks but SP have potentially incoherent references for which the compiler generates guarded accesses. Without the coherence protocol the usage of the hybrid memory system would not be possible in these cases, so the performance and energy consumption benefits it provides would not be exploited.

The reduction in execution time the hybrid memory system achieves when compared to a cache-based system can be observed in Figure 9. For each benchmark two bars are presented. The leftmost bar is the execution time of the cache-based system and the rightmost bar is the execution time of the hybrid memory system. Both bars are normalized to the cache-based system execution time and show the weight of each execution phase, considering as work time the whole execution time of the cache-based system. All benchmarks but EP present some degree of reduction. The reductions are mainly due to the reduction of execution time of the work phase, more than 35% in all cases. This big reduction in the work phase is caused by the better management of memory references in the hybrid memory system. First, the irregular accesses that reuse data along the execution of the benchmarks have a much higher L1 hit ratio in the hybrid memory system. This is because the hybrid memory system uses the LM to serve the regular accesses and the L1 to serve the irregular ones, so the data placed in the L1 is much less often evicted than in the cache-based system, where every access is served by the L1 so the data brought for irregular accesses is evicted when new data needs to be brought for regular references, causing misses when irregular accesses reuse data. The second important observation is that the hybrid memory system imposes an execution model that does extra work in the control and synchronization phases, but in the work phase it is able to execute the strided accesses without cache misses, since they are served by the LM. In the cache-based system, when a lot of strided memory references are being used, they cause collisions in the history tables of the prefetchers and also the big amount of prefetched data causes conflict misses in the whole cache hierarchy. These two situations are reflected in the AMAT and the L1 hit ratios shown in Table 3. MG and SP show a very similar behaviour, with respective reductions of 39% and 40% (or speedups of 1.64x and 1.66x). The big amount of regular references they have provoke conflict misses and collisions in the prefetchers in the cache-based system, which cause important penalties compared to the execution time spent in control phases in the hybrid memory system. CG, FT and IS show reductions of 26%, 24% and 36% (or speedups of 1.34x, 1.30x and 1.55x), respectively. These loops have fewer strided references but their critical path contains a potentially incoherent access with a high degree of reuse. These memory references almost always miss in the L1 in the cache-based system, while they are served very efficiently in the hybrid memory system. EP presents no speedup at all. In both architectures all accesses are served very efficiently, with similar AMATs and L1 hit ratios of 99.9% and 98.9%. An irregular store causes this difference in the hit ratio but

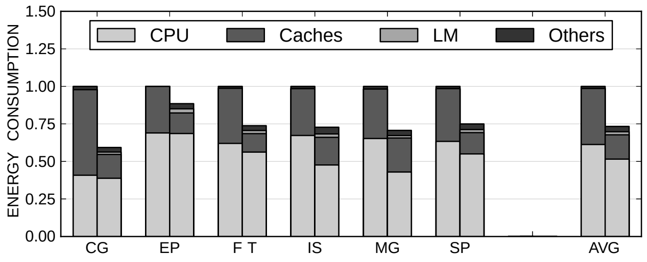


Fig. 10: Reduction in energy consumption.

this access is not in the critical path, so the difference in performance is 1%. On average, the speedup in all benchmarks is 1.38x, or a reduction in execution time of 28%.

Figure 10 shows, for each benchmark, the energy consumption of the cache-based system in the leftmost bar and of the hybrid memory system in the rightmost bar. Both bars are normalized to the cache-based system energy consumption and show the weight of each component of the processor on the total consumption. The energy components comprise the whole microarchiture of the core (CPU), the three levels of the cache hierarchy (Caches), the LM (LM) and the prefetchers, the DMA controller and the buses that connect all these components (Others). All benchmarks show reductions in energy consumption of 41% to 12%. In IS, MG and SP important savings come from the CPU. This is provoked by the reduction of cache misses, which cause energy penalties in the pipeline in the form of re-executed instructions. All benchmarks present energy reductions in the cache hierarchy, being CG and FT the ones that achieve the highest benefits. The energy consumed in the cache hierarchy decreases in all cases because, first, the hybrid memory system does fewer accesses to all the levels of the hierarchy because it uses the LM instead and, second, cache misses and data prefetches are more frequent in the cache-based system, provoking energy consumption due to cache line lookups and placements. This number of saved accesses is much larger than the activity that the hybrid memory systems provokes in the caches in the form of cache line lookups and invalidations due to the DMA transfers. All together, the resulting number of accesses to any level of the cache hierarchy decreases with the hybrid memory system, as can be observed in Table 3. Furthermore, the energy savings in the cache hierarchy are much bigger than the energy consumed by the LM in the hybrid memory system, which has a weight of less than 5%, and by the DMA engine, which also has a weight of less than 5%. The average savings in energy consumption in the benchmarks is 27%.

In conclusion, the hybrid memory system outperforms cache-based systems because it serves data very efficiently: the strided accesses are served by the LM so the cache hierarchy is less frequently accessed and it can be devoted to the data accessed by irregular and potentially incoherent accesses, avoiding evictions of data that is going to be reused. Moreover, fewer collisions in the history tables of the prefetchers happen due to the lower activity in the caches. This lower activity directly translates to less energy consumption, that is complemented with energy savings in the CPU due to the reduction of re-executed instructions caused by cache misses.

5 Related work

The idea of adding a LM alongside the cache hierarchy is not novel. This organization is found in commercial products like the NVIDIA Fermi [6]. In this platform the global memory (that is cached) and the LM are incoherent, and the architecture does not provide any mechanism to solve the coherence problem between the two storages. Instead, it relies on the programmer to explicitly manage the two memories. CUDA [34] provides keywords for the declaration of the variables to specify which memory will store them, so data replication does not happen. If two copies of data exist it is the programmer who has to explicitly declare and manage them, since neither the hardware nor the compiler give any support for coherence management between both memories.

Bertran et al. [12] propose to add a LM alongside the cache hierarchy in general purpose cores, but they do not solve the coherence problem between the two storages. Instead, they give the compiler the responsibility to discard loop transformations in case of coherence problems, restricting the effective utilization of the hybrid memory system.

Some works [35], [13] propose memory organizations that can be configured as caches, LMs or a combination of both. With such approaches, when the memory is logically configured as a hybrid memory system, the resulting system encounters the same coherence problem that this paper solves. The authors of Virtual Local Memories [13] allow to configure a part of the cache as a LM. When they do so they reserve for the LM a portion of the virtual address space that is direct-mapped to the physical address space and they offer the programmer a high-level API to move data between the LM and the SM with a DMA engine, ending up with an scheme that is identical to the one proposed in this paper. The authors of that work bypass the coherence problem by leaving the responsibility of managing the copies of data to the programmer. The coherence protocol for the hybrid memory system could be directly applied to their proposal to allow the compiler to generate code that manages the Virtual Local Memory. The memory hierarchy of the Smart Memories Architecture [35] also has the possibility to be configured as a combination of LM and caches. The authors focus on the hardware details that allow the configurability, but do not mention how the resulting configuration would be exposed to the upper layers of the system. If the Smart Memories Architecture adopted the same scheme that the hybrid memory system and the Virtual Local Memories assume, the proposed coherence protocol could also be directly applied to that work.

Cohesion [36] allows the software to dynamically select which cache lines are cache coherent by enabling and disabling the cache coherence protocol for specific lines. This approach faces the same problem as the hybrid memory system because it opens the door to incoherent copies of data, relying on the programmer to explicitly manage them.

This paper relies on previous works on DMA coherence [15]. The IBM Cell architecture [5], [14] ensures DMA coherence by doing lookups in the cache hierarchy when DMA transfers are performed. In the Cell architecture only DMA transfers can generate data replication and there are no coherence problems because, with regular memory instructions, the accelerator cores can only access their LMs and the general purpose core can only access the cache hierarchy. Whenever a modification has to be visible to other cores DMAs are used so the coherence is ensured. In the hybrid memory system this approach is extended to support coherence at the memory instruction level because a core can access both memories.

D. Tang et al. [37] introduce on-chip storage to separate IO data from CPU data. Although with different motivations, this work faces similar coherence problems as the ones the proposed coherence protocol addresses. The introduction of the DMA-cache creates potential incoherences that are solved by a refinement of the MOESI and ESI cache coherence protocols. In the coherent hybrid memory system data invalidation only happens along a *dma-put* and never a memory access to the cache hierarchy can modify the contents of the LM.

6 Conclusions

The hybrid memory system, which consists of adding a local memory alongside the cache hierarchy, is a promising solution to the lack of scalability and the power consumption problems of future cache coherent multicore and manycore architectures. One of the main problems of the hybrid memory system is the incoherence between the two storages, for which this paper proposes a novel hardware/software coherence protocol.

The protocol admits data replication in the two storages and avoids keeping them coherent. Instead, it ensures that the valid copy of the data is always accessed. The design consists of a hardware directory that keeps track of the contents of the local memory and guarded memory instructions that the compiler selectively emits for potentially incoherent memory accesses. Guarded instructions access the directory and then are diverted to the storage where the correct copy of the data is. The main achievement of the coherence protocol is that the compiler algorithm to generate code for the hybrid memory system is straightforward and always safe because it is not limited by memory aliasing problems.

The proposed coherence protocol introduces average overheads of 0.26% in execution time and of 2.03% in energy consumption to enable the usage of the hybrid memory system. This system, compared to a cache-based system, provides an average speedup of 38% and an energy reduction of 27%.

Acknowledgments

We thankfully acknowledge the support of the the Spanish Ministry of Education (TIN2007-60625 and CSD2007-00050), the Generalitat de Catalunya (2009-SGR-980), the HiPEAC Network of Excellence (contracts EU FP7/ICT 217068 and 287759), and the BSC-IBM collaboration agreement.

References

[1] J. Leverich, H. Arakida, A. Solomatnikov, A. Firoozshahian, M. Horowitz, and C. Kozyrakis, "Comparing Memory Systems for Chip Multiprocessors," *SIGARCH Computer Architecture News*, pp. 358–368, 2007.

[2] R. Murphy, "On the Effects of Memory Latency and Bandwidth on Supercomputer Application Performance," in *IISWC '07: Proceedings of the 10th International Symposium on Workload Characterization*. IEEE Computer Society, 2007, pp. 35–43.

[3] A. Ros, M. E. Acacio, and J. M. García, *Parallel and Distributing Computing*. IN-TECH, 2010, ch. Cache Coherence Protocols for Many-Core CMPs.

[4] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad Memory: A Design Alternative for Cache On-chip Memory in Embedded Systems," in *CODES '02: Proceedings of the 10th International Symposium on Hardware/Software Codesign*. ACM, 2002, pp. 73–78.

[5] J. Kahle, "The Cell Processor Architecture," in *MICRO 38: Proceedings of the 38th International Symposium on Microarchitecture*. IEEE Computer Society, 2005, pp. 3–4.

[6] P. N. Glaskowsky, "NVIDIA's Fermi: The First Complete GPU Computing Architecture." White paper, 2009.

[7] 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 *PACT '08: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. ACM, 2008, pp. 292–302.

[8] W. Landi and B. G. Ryder, "A Safe Approximate Algorithm for Interprocedural Aliasing," in *PLDI '92: Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*. ACM, 1992, pp. 473–489.

[9] A. Deutsch, "Interprocedural May-Alias Analysis for Pointers: Beyond k-limiting," in *PLDI '94: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*. ACM, 1994, pp. 230–241.

[10] R. P. Wilson and M. S. Lam, "Efficient Context-Sensitive Pointer Analysis for C Programs," in *PLDI '95: Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*. ACM, 1995, pp. 1–12.

[11] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann, 2002.

[12] R. Bertran, M. Gonzàlez, X. Martorell, N. Navarro, and E. Ayguadé, "Local Memory Design Space Exploration for High-Performance Computing," *The Computer Journal*, pp. 786–799, 2010.

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

[14] M. Kistler, M. Perrone, and F. Petrini, "Cell Multiprocessor Communication Network: Built for Speed," *IEEE Micro*, pp. 10–23, 2006.

[15] T. B. Berg, "Maintaining I/O Data Coherence in Embedded Multicore Systems," *IEEE Micro*, pp. 10–19, 2009.

[16] "MPI: A Message-Passing Interface Standard. 2003."

[17] "OpenMP Application Program Interface. Version 3.0. May 2008."

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

[19] 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^{TM}Architecture," *IBM Systems Journal*, pp. 59–84, 2006.

[20] 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 *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society, 2005, pp. 161–172.

[21] Y. Paek, J. Hoeflinger, and D. Padua, "Efficient and Precise Array Access Analysis," *ACM Transactions on Programming Languages and Systems*, pp. 65–109, 2002.

[22] T. Chen, T. Zhang, Z. Sura, and M. G. Tallada, "Prefetching Irregular References for Software Cache on Cell," in *CGO '08: Proceedings of the 6th International Symposium on Code Generation and Optimization*. ACM, 2008, pp. 155–164.

[23] "Power ISA. Version 2.06 Revision B. IBM. July 2010."

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

[25] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. CACTI 6.0: A Tool to Understand Large Caches. 2009.

[26] R. C. Murphy and P. M. Kogge, "On the Memory Access Patterns of Supercomputer Applications: Benchmark Selection and Its Implications," *IEEE Transactions on Computers*, pp. 937–945, 2007.

[27] J. Weinberg, M. O. McCracken, E. Strohmaier, and A. Snavely, "Quantifying Locality In The Memory Access Patterns of HPC Applications," in *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 2005, pp. 50–62.

[28] M. T. Yourst, "PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator," in *ISPASS '07: Proceedings of the 7th International Symposium on Performance Analysis of Systems and Software*. IEEE Computer Society, 2007, pp. 23–34.

[29] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," in *ISCA '00: Proceedings of the 27th International Symposium on Computer architecture*. ACM, 2000, pp. 83–94.

[30] T.-F. Chen and J.-L. Baer, "Effective Hardware-Based Data Prefetching for High-performance Processors," *IEEE Transactions on Computers*, pp. 609–623, 1995.

[31] J. Doweck, "Inside Intel Core Microarchitecture and Smart Memory Access. An In-Depth Look at Intel Innovations for Accelerating Execution of Memory-Related Instructions." White paper, 2006.

[32] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS Parallel Benchmarks," in *SC '91: Proceedings of the 1991 Conference on Supercomputing*. IEEE Computer Society, 1991, pp. 158–165.

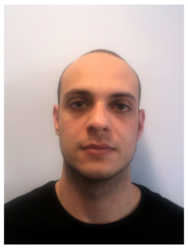
[33] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically Characterizing Large Scale Program Behavior," in *ASPLOS '02: Proceedings of the 10th nternational conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2002, pp. 45–57.

[34] "NVIDIA CUDA C Programming Guide. Version 4.2. April 2012."

[35] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz, "Smart Memories: A Modular Reconfigurable Architecture," in *ISCA '00: Proceedings of the 27th International Symposium on Computer architecture*. ACM, 2000, pp. 161–171.

[36] J. H. Kelm, D. R. Johnson, W. Tuohy, S. S. Lumetta, and S. J. Patel, "Cohesion: An Adaptive Hybrid Memory Model for Accelerators," *IEEE Micro*, pp. 42–55, 2011.

[37] D. Tang, Y. Bao, W. Hu, and M. Chen, "DMA Cache: Using On-Chip Storage to Architecturally Separate I/O Data from CPU Data for Improving I/O Performance," in *HPCA '10: Proceedings of the 16th International Conference on High-Performance Computer Architecture*. IEEE Computer Society, 2010, pp. 1–12.



Lluc Alvarez received a bachelor's degree in Computer Systems from Universitat de les Illes Balears in 2006 and a master's degree in Computer Architecture from Universitat Politècnica de Catalunya (UPC) in 2009. Since 2010 he is a PhD student in the Computer Architecture Department at UPC and a resident student at Barcelona Supercomputing Center. His main research interests are computer microarchitecture and memory hierarchies of multicore architectures for high-performance computing.

Lluís Vilanova is a PhD student at the Barcelona Supercomputing Center and the Computer Architecture Department at the Universitat Politècnica de Catalunya, from where he also received his bachelor's degree in Computer Science in 2006 and his master's degree in Computer Architecture in 2008. His interests cover computer architecture and operating systems.

Marc Gonzàlez received the Engineering degree in Computer Science in 1996 and the Computer Science PhD degree on December 2003. He currently holds an Associate Professor position in the Computer Architecture Department from the Technical University of Catalonia. His research activity is linked to the Barcelona Supercomputing Center (BSC) as a collaborator. His main interests are both parallel programming and computer architecture, specifically for hybrid multi-core systems. Besides. he has worked on power and energy modeling techniques for multi-core processors and on parallel programming models, with special interest in the OpenMP and OpenCL paradigms. Up today, he has published more than 40 refereed papers in journals and conferences.

Xavier Martorell received the M.S. and Ph.D. degrees in Computer Science from the Technical University of Catalunya (UPC) in 1991 and 1999, respectively. He has been an associate professor in the Computer Architecture Department at UPC since 2001, teaching on operating systems. His research interests cover the areas of paralellism, runtime systems, compilers and applications for high-performance multiprocessor systems. Since 2005 he is the manager of the team working on Parallel Programming Models at the Barcelona Supercomputing Center. He has participated in several european projects dealing with parallel environments (Nanos, Intone, POP, SARC, ACOTES). He is currently participating in the European HiPEAC2 Network of Excellence, and the ENCORE european project.

Nacho Navarro is Associate Professor at the Universitat Politecnica de Catalunya (UPC), Barcelona, Spain, since 1994, and Senior Researcher at the Barcelona Supercomputing Center (BSC), serving as manager of the Accelerators for High Performance Computing group. He holds a Ph.D. degree in Computer Science from UPC. His current interests include: GPGPU computing, multi-core computer architectures, hardware accelerators, dynamic reconfigurable logic support, memory management and runtime optimizations. He is also doing research on massively parallel computing at the University of Illinois (IMPACT Research Group). Prof. Navarro is a member of IEEE, the IEEE Computer Society, the ACM and the HiPEAC NOE.

Eduard Ayguadé received the Engineering degree in Telecommunications in 1986 and the Ph.D. degree in Computer Science in 1989, both from the Universitat Politècnica de Catalunya (UPC), Spain. Since 1987 he has been lecturing on computer organization and architecture and parallel programming models. Currently, and since 1997, he is full professor of the Computer Architecture Department at UPC. His research interests cover the areas of processor microarchitecture, multicore architectures and programming models and their architectural support. He has published more than 100 papers in these topics and participated in several research projects in the framework of the European Union and research collaborations with companies. He is associated director for research on computer sciences at the Barcelona Supercomputing Center (BSC-CNS).