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

Lluc Alvarez*†, Lluís Vilanova*†, Marc Gonzalez*†, Xavier Martorell*†, Nacho Navarro*†, Eduard Ayguade*†

*Barcelona Supercomputing Center
C. Jordi Girona, 29
08034 Barcelona, Spain
Email: name.surname@bsc.es

†Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
C. Jordi Girona, 1-3
08034 Barcelona, Spain

***Abstract*—Cache coherence protocols limit the scalability of chip multiprocessors. One solution is to introduce a local memory alongside the cache hierarchy, forming a hybrid memory system. Local memories are more power-efficient than caches and they 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 incoherency 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. Coherency is ensured by a simple 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 0.24% in execution time and of 1.06% in energy consumption to enable the usage of the hybrid memory system.**

I. Introduction

Upcoming chip multiprocessor (CMP) architectures are expected to include a significant number of cores, as a result of the replication of general purpose and specialized accelerator cores. As an immediate consequence, the memory subsystem has to evolve into some novel organization that satisfies the inherent bandwidth requirements of such approach and avoids potential bottlenecks in the shared levels of the memory hierarchy. Both the power consumption originated in the memory hierarchy and the lack of scalability of current cache coherence protocols 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 any 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 regular 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 being currently used in GPGPUs [6] and in general purpose cores [7].

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 [8] but, when non-predictable memory access patterns are found, compilers need to ensure correctness by applying complex analyses such as memory aliasing and data flow analysis [9], [10], [11]. When compilers cannot ensure that there is not going to be 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 is caused by the fact that 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, that are diverted to the correct copy of the data at execution time. 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 overheads of 0.24% in execution time and 1.06% 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 speedup of 36% and an energy saving of 18% when compared to a cache-based system.

The rest of this paper is organized as follows: Section II gives some background of how a LM is integrated in a core and how the resulting architecture is programmed. Section III explains the design of the coherence protocol and Section IV presents its evaluation. Section V comments some related work and Section VI remarks the main conclusions of this work.

SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00 ©2012 IEEE

II. Background and Motivation

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

A. 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. The system reserves a range of physical addresses for the LM and these physical addresses are direct-mapped from a virtual memory range, which is identified by two registers that specify the base virtual address and the LM size. Thus, the CPU is able to access the LM using regular loads and stores to these virtual addresses. 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 [12] 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 [7]. This scheme has two important benefits. First, the access time to the LM is constant because no pagination is needed. Second, it allows the introduction of the LM in a very simple way because only two extra registers are required to configure the LM and there is no interference with the cache hierarchy.

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 registers in the DMAC.

DMA transfers are coherent with the SM [13], [14] 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. In a typical configuration with private write-through L1 caches, private write-back L2 caches and a shared L3 cache the implementation of coherent DMA transfers depends on the CMP cache coherence protocol. The ideal case is to have a directory-based protocol, so it is straightforward to access the cache that keeps the valid copy of the data. Contrariwise, with a snoop-based protocol this is very costly, since in a CMP every bus request performs a lookup in the L2 cache of every core, generating a huge overhead in energy consumption. This situation can be alleviated by changing the write policy of the L2 caches to write-through, so the bus requests only do one lookup in the shared L3 cache. Using write-through L2 caches is inefficient in cache-based systems because every store reaches the L3 cache, adding a big overhead in energy consumption and bus traffic. In the hybrid memory system this is less critical, since most accesses are served by the LM so, even with write-through L2 caches, the number of accesses to any level of the cache hierarchy decreases when compared to a cache-based system with write-back L2 caches. In this paper the hybrid memory system uses a snoop-based coherence protocol with write-through L2 caches to show that, even in this worst case scenario, it provides many benefits compared to cache-based systems.

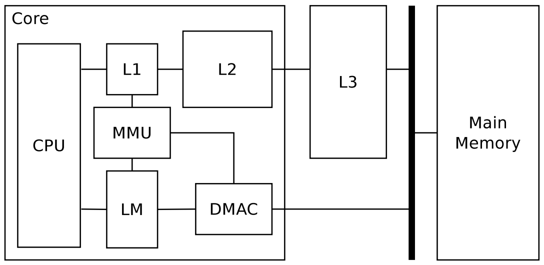


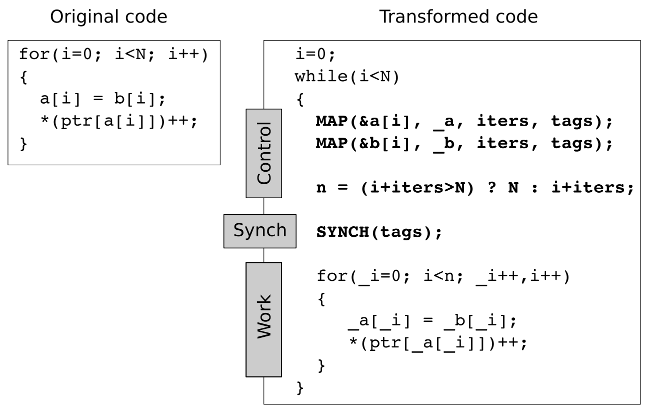
Fig. 1: Architecture of the hybrid memory system.

B. Programming Model

A CMP that uses the hybrid memory system can be programmed using any programming model for cache-based CMPs. The only thing to be done for the hybrid memory system is to map private data to the LM. Typically programming models rely on programmers to know how the data of a parallel program has to be distributed. In distributed memory architectures, programming models such as MPI [15] 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 data partitioning. In OpenMP [16] 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 intrinsics of the programming models themselves.

The data assigned to each core is then mapped to its LM, inducing a particular execution model. The working set is split in blocks, as the total amount of data typically exceeds the size of the LM, and these blocks are explicitly moved between the LM and the SM. In the case of a computational loop, this is accomplished by converting the code into a two-level nested iterative structure, as shown in Figure 2. Each iteration of the outermost loop has three phases: (1) a control phase where data is moved between the LM and the SM, (2) a synchronization phase to wait for the DMA transfers to finish and (3) a work phase where the computation for the current block is performed. These code transformations are usually done by run-time libraries [8], [17] or compilers [18], [19].

Automatic code transformations decide which data is mapped to the LM by analyzing the memory accesses [20]. *Regular accesses* are those that expose predictable access patterns (e.g., with a constant stride). These are mapped to the LM. Unpredictable memory accesses are difficult to map to the LM [8], so they are served by the cache hierarchy. These are called *irregular accesses*. In Figure 2, accesses to `a` and `b` are regular accesses, and the access to `ptr` is irregular.



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

Fig. 2: Code transformation and 3-phase execution model.

In the control phase, the data needed for regular accesses in the next work phase is brought to the LM (`MAP` primitive in Figure 2). Blocks of data are copied from the SM to the LM, potentially sending back to the SM some previously used blocks. Even in case of mapping a block of data to the LM for writing only, the transfer of the block from the SM to the LM is done because otherwise, if only part of the block was modified, the write-back to the SM would update the unmodified parts of the copy in the SM with garbage. In order to do these actions in a simple and efficient way fixed-size buffers are used. The compiler decides how many buffers will be needed to handle all regular accesses in the loop and, depending on the size of the LM, sets the size of the buffers and assigns them to fixed addresses in the LM. 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 one half of the storage.

The work phase is similar to 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 is determined by 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 (`ptr`).

C. 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. This problem arises because the compiler creates a copy of the data when maps it to the LM and, for regular accesses, it 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 [9], [10], [11]. In Figure 2 this means predicting when, if ever, one instance of the `ptr` access aliases with any instance of the `_a` and `_b` accesses. Current algorithms are not able to solve this problem in the general case, so compilers adopt restrictive solutions. The naive one is to discard using 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 [8], adding big overheads because DMA transfers of small sizes are inefficient. Software caching is another solution [21], [8]. These keep track of the contents of the LM with a software directory and do a costly lookup prior to every potentially incoherent access to decide if it goes to the LM or to the SM.

This paper proposes an efficient mechanism that ensures coherency 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.

III. 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 coherence protocol is independent of the CMP 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 CMP coherence protocol. 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 the copy of this data in the SM. This is key to ensure there is no interaction with the CMP 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 task of the compiler and the hardware support the coherence protocol requires.

A. 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 algorithm has three steps: classification of memory references, code transformation and code generation.

1) Classification of Memory References: The main task of the compiler in this phase is to identify which memory accesses are suitable to be mapped to the LM and which others to the SM. It does so by classifying the memory accesses according to their access patterns and possible aliasing hazards:

- *Regular accesses* are those that expose a strided access pattern. They are served by the LM.
- *Irregular accesses* are those that do not expose a strided access pattern and the compiler is sure there is no aliasing with the contents of the LM. They are served by the SM.
- *Potentially incoherent accesses* are those that do not expose a strided access pattern and the compiler is not sure there is no aliasing with the contents of the LM. They access the directory and then the SM or the LM.

2) Code Transformation: In this phase the compiler does the code transformations for regular accesses shown in Figure 2. These are typical transformations to manage LMs with software caches [19], [8]. The compiler also informs the hardware of the size of the LM buffers. For irregular and potentially incoherent accesses nothing is done in this phase.

3) Code Generation: In this phase the compiler generates memory instructions for the references:

- For *regular accesses* the compiler generates instructions that directly access the LM. This is accomplished by using addresses that are computed as a base address of a LM buffer plus an offset.
- For *irregular accesses* the compiler generates instructions that directly access the SM. This is accomplished by using addresses that are computed as a base address in SM plus an offset.
- For *potentially incoherent accesses* the compiler generates guarded instructions. The instruction is first generated in the same way as the instructions generated for *irregular accesses*, so an initial SM address is generated. Then the guard is inserted in the instruction, so it will access the directory using the SM address and will be diverted to the corresponding memory. The implementation of the guard is discussed later in this section.

One special case has to be treated separately. When the compiler determines a write access is potentially incoherent and it aliases with some data that is mapped for reading only, a guarded store is generated for it. The execution of the guarded store will hit in the directory and the write will be done to the LM. This may lead to an erroneous execution because, since the mapping to the LM is for reading only, no write-back of the data to SM will be programmed and, when the buffer is reused to map new data, the *dma-get* operation will overwrite the modifications done by the potentially incoherent store. This problem can be solved by making 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 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. Notice that if the lookup in the directory of the second 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 [12] 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.

The implementation of the guarded memory operations is highly dependent on the architecture. On a RISC architecture the ISA should be extended to duplicate all memory instructions with a guarded form. As this might produce many new opcodes, there are other alternatives. One solution is to take unused bits of the binary representation of memory instructions, as happens in PowerPC [22]. Another option is to provide a fewer range of guarded memory instructions and restrict the compiler to these. In CISC architectures like x86 [23], 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 will be consumed by the memory instruction, conceptually converting the guarded memory access to a coherency-aware address calculation plus a normal memory operation.

B. 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.

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.

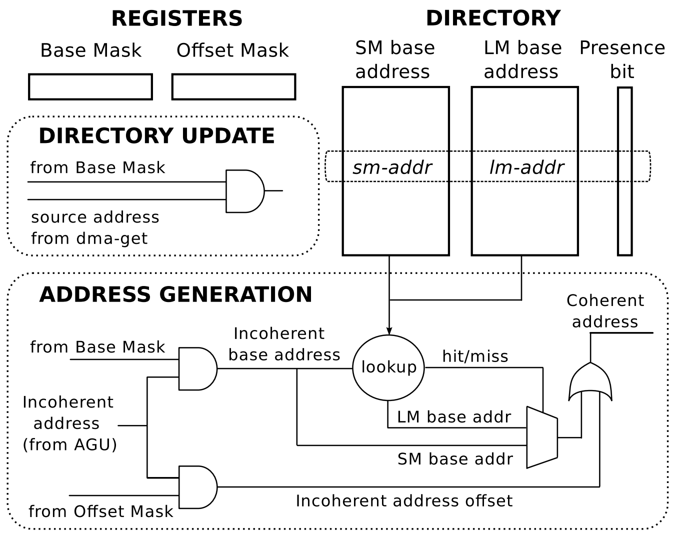


Fig. 3: Scheme and main operations of the directory.

Address generation: The directory is used in the address generation as shown in Figure 3. The Address Generation Unit (AGU) [12] generates a potentially incoherent SM address (*Incoherent address*) with the operands of the guarded instruction. 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 by 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 [24], with a process technology of 45nm, the latency of the directory is 0.348 nanoseconds. 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 the address generation in the AGU, which is an extremely simple operation, 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, which is not a 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 [25], [26], the directory is rarely accessed and the goodnesses of the hybrid memory system are preserved.

C. Data Coherency 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.

1) Data States and Operations: Figure 4 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 resident 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.

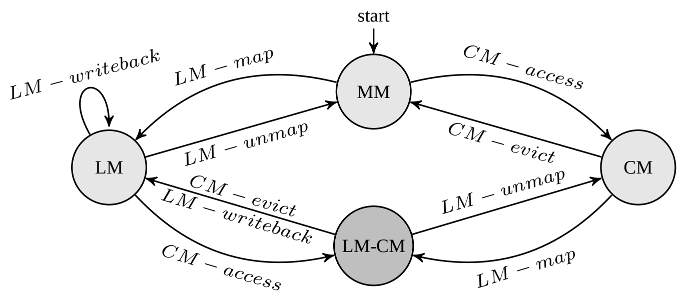


Fig. 4: State diagram of possible data replication states.

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 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.

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 coherency. 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 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 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→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.

IV. 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.

A. Experimental Framework

The proposal has been evaluated using PTLsim [27], extending it with a LM, a DMAC and the directory of the coherence protocol. For the energy results Wattch [28] has been embedded into the simulator. Cycle-accurate single-core simulations are presented because the coherence protocol is per core. Table I shows the parameters of the simulator.

Six typical HPC benchmarks from the NAS benchmark suite [31] are used for the evaluation. The benchmarks have been compiled using GCC 4.6.1 with the -O3 optimization flag on. SimPoint [32] has been used to identify the simulation points and at least 150 millions of x86 instructions have been simulated for each benchmark.

To generate guarded memory instructions for the potentially incoherent accesses it has been checked the outcome of the

TABLE I: 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, 2 cycles latency
L2 cache	256 KB, 24-way set-associative, 15 cycles latency
L3 cache	4 MB, 32-way set-associative, 40 cycles latency
Prefetcher	IP-based stream prefetcher [29], [30] to L1, L2, L3
Local memory	32 KB, 2 cycles latency

alias analysis performed by GCC on every memory reference. 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. Instruction prefixes are used to implement the guards as explained in Section III-A.

B. Overhead of the Coherence Protocol

A microbenchmark that stresses the coherence protocol is used to facilitate the study of its performance overheads. Table II shows its characteristics. The microbenchmark consists of 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]` may alias, so a guarded load is generated. The guard is represented in bold font in the assembly code. In the WR mode it is assumed 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 generated. The RD/WR mode is a combination of the RD mode and the WR mode. In addition, it is possible to adjust the percentage of memory operations that need to be guarded in order to model all possible scenarios in terms of the ratio of accesses that are potentially incoherent.

Figure 5 shows the performance overhead 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 overheads are computed against the baseline mode.

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 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

TABLE II: Scheme of the microbenchmark.

Microbenchmark	Mode	Assembly code
`int a[N];` `int c;` `for(i=0; i<N-1; i++) {` `a[i+1] = a[i] + c;` `}`	Baseline	`mov a(,esi,4),ebx`
		`add 0x0(c),ebx`
		`mov ebx,a+4(,esi,4)`
	RD	**`mov a(,esi,4),ebx`**
		`add 0x0(c),ebx`
		`mov ebx,a+4(,esi,4)`
	WR	`mov a(,esi,4),ebx`
		`add 0x0(c),ebx`
		`mov ebx,a+4(,esi,4)`
		`mov ebx,a+4(,esi,4)`
	RD/WR	**`mov a(,esi,4),ebx`**
		`add 0x0(c),ebx`
		`mov ebx,a+4(,esi,4)`
		`mov ebx,a+4(,esi,4)`

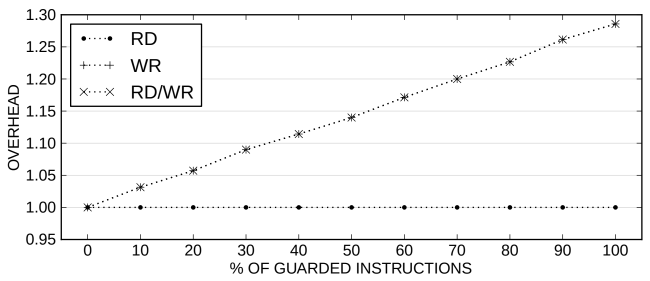


Fig. 5: Overhead in all microbenchmark modes.

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 6 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 0.99% and 0.43%, 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

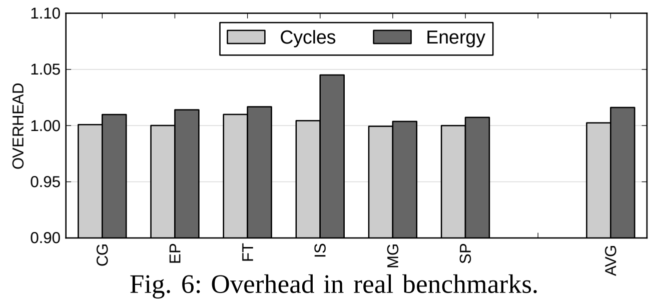


Fig. 6: Overhead in real benchmarks.

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 executed instructions. This increase in instructions barely affects 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 latency of 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, 0.24%.

Figure 6 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 4.5%. The overhead generated by the directory is around 1.5%, the remaining 3% is caused by the execution of the double store. The average overhead in energy consumption of all benchmarks is 1.06%.

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.

C. 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.

This section compares the coherent hybrid memory system with a cache-based system. The two architectures have exactly the same characteristics but with two differences. First, 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. Second, the write policy of the L2 is write-through in the hybrid memory system and write-back in the cache-based system. Table III 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, the number of write-throughs or write-backs that are performed from the L2 to the L3 (L3 WT/WB), 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 III 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 7. 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 34% 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

TABLE III: 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	WT/WB	Accesses	Accesses	Accesses
CG	Hybrid coherent	1/7 (14%)	3.15	90.52	19319	26376	2388	14973	30235	10566
CG	Cache-based	0	4.31	82.23	70371	62822	8654	84202	0	0
EP	Hybrid coherent	1/20 (5%)	2.14	99.93	37152	10266	10069	10589	3862	3519
EP	Cache-based	0	2.37	98.93	43814	13219	166	797	0	0
FT	Hybrid coherent	4/34 (11%)	2.60	96.61	912779	761009	654686	879602	1155150	55118
FT	Cache-based	0	4.95	78.54	1379688	789765	69253	352269	0	0
IS	Hybrid coherent	2/5 (25%)	6.27	74.00	140663	194465	99344	168968	73400	25714
IS	Cache-based	0	7.93	64.10	169425	182716	34315	127692	0	0
MG	Hybrid coherent	1/60 (1.66%)	2.24	99.71	605269	252799	237447	281129	798562	19377
MG	Cache-based	0	3.89	90.65	827239	238099	19783	127176	0	0
SP	Hybrid coherent	0/497 (0%)	2.41	98.37	331832	162441	149649	174211	235024	0
SP	Cache-based	0	4.73	79.59	407952	164515	11866	82301	0	0

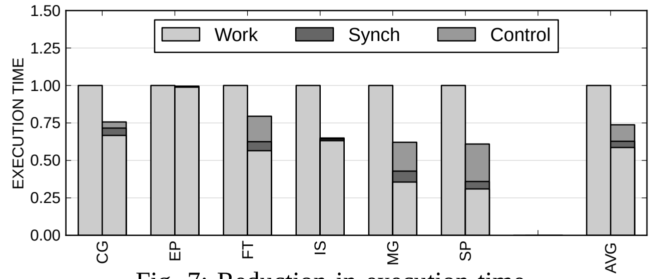


Fig. 7: Reduction in execution time.

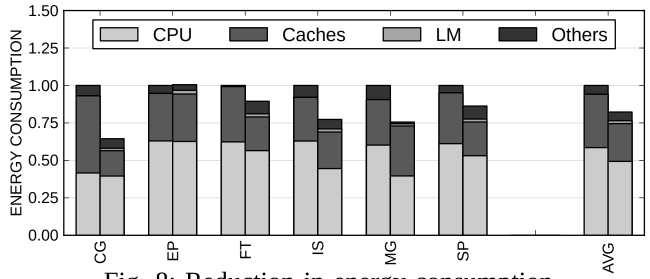


Fig. 8: Reduction in energy consumption.

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 III. MG and SP show a very similar behaviour, with respective reductions of 38% and 40% (or speedups of 1.61x and 1.64x). The big amount of regular references they have provoke conflict misses and collisions in the prefetchers in the cache-based system, that cause very important penalties compared to the execution time spent in control phases in the hybrid memory system. CG, IS and FT show reductions of 25%, 35% and 21% (or speedups of 1.32x, 1.25x and 1.54x), 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 and this access is not in the critical path, so the differences in performance are less than 0.5%. On average, the speedup in all benchmarks is 1.36x, or a reduction of 27%.

Figure 8 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. All benchmarks but EP show reductions in energy consumption of 36% to 11%. In IS, MG and SP the major 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. Other benchmarks like CG and FT present important energy reductions in the cache hierarchy. This is because, first, the hybrid memory system does fewer accesses to the L1 and L2 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. These numbers can be observed in Table III. The number of accesses to the L3 increases due to the write-through policy in the L2, but this increase is lower than the savings in the rest of the hierarchy and so energy savings are achieved. Furthermore, the energy savings in the cache hierarchy are bigger than the energy consumed by the LM in the hybrid memory system, which has a weight of less than 5%. Unlike the rest of benchmarks, EP has an overhead of less than 1%. In the hybrid memory system the strided accesses are served by the LM, but every store to the L1 reaches the L3 due to the write-through policy in the L2. The gain and the loss compensate one with each other to result in roughly the same energy consumption. The average savings in energy consumption in the benchmarks is 18%.

It is important to show that it is possible to have write-through L2 caches in the hybrid memory system. This configuration is the worst case scenario to implement coherence at the DMA transfer level in a CMP, as explained in Section II-A. Table III shows it is viable to have write-through L2 caches in this architecture. The column L3 WT/WB shows the number of accesses to the L3 that are provoked by write-throughs from the L2 in the hybrid memory system and by write-backs from the L2 in the cache-based system. The best case for the hybrid memory system is to use the cache hierarchy for irregular read references only and serve all the stores with the LM, like in CG. In this case the number of write-throughs is lower than the number of write-backs done by the cache-based system, resulting in a 6 times lower number of accesses to the L3. When the cache hierarchy is used only to serve irregular write references (IS) the increase in L3 accesses is 33%. The worst case is when there are a lot of stores to local variables, either in the computational kernel (EP) or because of the complexity of the control phases due to the big amount of strided references (FT, MG and SP). The number of accesses to the L3 grows by factors between 5x to two orders of magnitude, but this increase is balanced by the reduction of accesses to the L1 and the L2. In addition, solutions like a hardware stack or mapping the stack to the LM could eliminate this problem.

In conclusion, the hybrid memory system outperforms cache-based systems because it is able to serve data very efficiently: the strided accesses are served by the LM so the first levels of the cache hierarchy are less frequently accessed and their capacity 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 first cache levels. This lower activity directly translates to less energy consumption, being the major savings due to the reduction of re-executed instructions caused by cache misses. The energy savings in the L1 and the L2 caches are very important also, but are almost compensated by the increase in the energy consumed in the L3. This increase in the L3 is caused by the write-through policy of the L2 cache in the hybrid memory system, that generates an important amount of accesses to the L3 when a lot of local variables are used.

V. Related work

The idea of adding a LM alongside the cache hierarchy is not novel. Bertran et al. [7] propose such organization in a general purpose core, and it is also present in commercial products like the NVIDIA Fermi GPGPU [6]. These two approaches do not solve the coherence problem between the two storages. Bertran et al. [7] give the compiler the responsibility to discard loop transformations in case of coherence problems, restricting the effective utilization of the hybrid memory system. In the NVIDIA Fermi [6] the global memory (that is cached) and the LM are incoherent, and it is the programmer who explicitly manages them. The programming language provides keywords that are used in the declaration of the variables to specify which memory will store them, so data replication does not happen. If two copies of data exist in the two memories it is the programmer who has to explicitly declare and manage them, since neither the hardware nor the compiler give any support for coherency management between both memories.

Cohesion [33] 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 [14]. The IBM Cell architecture [5], [13] 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. [34] 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.

VI. Conclusions

The hybrid memory system, which consists of adding a local memory alongside the cache hierarchy, can be a solution to the lack scalability of future CMPs. One of the main problems of such approach is the lack of coherence between the two storages. The main contribution of this paper is the design of a coherence protocol for hybrid memory systems.

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. This coherence protocol has important goodnesses. The task of the compiler when it generates code for the hybrid memory system becomes straightforward and is always safe because it is not limited by memory aliasing problems. Furthermore, the coherence protocol can be implemented with simple hardware additions and the design does not introduce any overheads in codes that do not present coherence problems.

The evaluation shows the coherence protocol introduces overheads of 0.24% in execution time and of 1.06% in energy consumption to enable the usage of the hybrid memory system, that achieves an average speedup of 36% and an energy reduction of 18% when compared to a cache-based system.

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] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

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

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

[15] “MPI: A Message-Passing Interface Standard. 2003.”

[16] “OpenMP Application Program Interface. Version 3.0. May 2008.”

[17] 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.

[18] 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.

[19] 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.

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

[21] 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.

[22] “Power ISA. Version 2.06 Revision B. IBM. July 2010.”

[23] “Intel 64 and IA-32 Architectures Software Developer’s Manual. January 2011.”

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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

[30] 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.

[31] 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.

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

[33] 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.

[34] 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.