Hardware/Software Coherence in Hybrid Memory Models

Lluc Alvarez*†, Nikola Vujic*†, Lluis Vilanova*†, Ramon Bertran*†,
Marc Gonzalez*†, Xavier Martorell*†, Nacho Navarro*† and Eduard Ayguade*†
*Barcelona Supercomputing Center
Jordi Girona 29, 08034 Barcelona, Spain,
Email:{lluc.alvarez, nikola.vujic, lluis.vilanova, ramon.bertran, xavier.martorell, eduard.ayguade}@bsc.es
†Universitat Politecnica de Catalunya
Jordi Girona 1-3, 08034 Barcelona, Spain
Email: {marc, nacho}@ac.upc.edu

***Abstract*—Current cache coherence protocols limit the scalability of chip multiprocessor (CMP) architectures. The expected increase of the number of cores in next generation CMPs call for an evolution of the memory subsystem. One solution is to introduce a local memory side to the cache hierarchy, forming a hybrid memory model. On the one hand, local memories are more power-efficient than caches and they don't generate coherence traffic. On the other hand, local memories suffer from poor programmability, so programmers rely on automatic code transformations to operate them. When non-predictable memory access patterns are found compilers do not succeed in generating code that manages the local memory because they require complex memory aliasing analyses. This is caused by the incoherency between the local memory and the cache hierarchy. This paper proposes a coherence protocol for hybrid memory models that allows the compiler to generate code even in the presence of aliasing problems. Coherency is ensured by a simple software/hardware co-design that identifies potentially incoherent memory accesses and diverts them to the correct copy of the data. The coherence protocol doesn't maintain two coherent copies of the data, so no coherence traffic is generated and the overhead is negligible, 0.2% on average. When compared to traditional cache-based architectures, the hybrid memory model with the proposed coherence protocol achieves an average speedup of 1.5x.**

I. Introduction

Next generation 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 up to 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 memory coherence protocols constrain the sharing and the size of caches when cores are replicated up to a certain level [1]–[4].

One possible solution to the lack of scalability of current coherence protocols is the introduction of on-chip local memories, also known as scratchpad memories [5]. Local memories have been successfully introduced in the high performance computing (HPC) domain in several ways. In the Cell architecture [6], accelerator cores access their private local memory with regular memory instructions and use explicit DMA transfers to move data between memories. A more recent trend is to introduce a local memory side to the cache hierarchy, forming a hybrid memory model. This hybrid approach is being currently used in GPGPUs [7] and in general purpose cores [8]. The main advantages of local memories are that they offer deterministic 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 local memories introduce programmability difficulties. They are easy to manage when the computation is based on predictable memory access patterns [9] but, when non-predictable memory access patterns are found, they require complex compiler analyses such as memory aliasing and data flow analysis [10]–[12]. When the compiler cannot ensure that there is not going to be aliasing between two memory references that may target copies of the same data in the local memory and in the cache hierarchy it must conservatively avoid using the local memory. This problem is caused by the fact that the copies of data in the local memory and in the cache hierarchy are incoherent.

The main contribution of this paper is a novel coherent hybrid memory model that exploits compiler and hardware co-design to achieve the programmability of a pure cache-based system by safely enabling the use of the local memory even under the presence of aliasing problems. A coherent memory view between the two storages is ensured by a simple hardware/software coordinated mechanism, implemented by two key components: (1) a lightweight per-core hardware directory that keeps track of which data is resident in the local memory and (2) guard instructions for memory operations that the compiler selectively places in potentially incoherent data accesses. The resulting design allows the compiler to use a straightforward algorithm to generate code for the hybrid memory model, operating the local memory as a software-managed cache unified with the existing hardware cache hierarchy. The evaluation of

the coherence protocol shows that it introduces an average overhead in execution time of 0.2% in real benchmarks, being zero in most cases and never exceeding a 3.35%. When compared to an architecture based exclusively on caches, the coherent hybrid memory model achieves an average speedup of 1.5x.

The rest of this paper is organized in the following sections. Section II gives some background of how a local memory is integrated in a general purpose core and how the resulting architecture is programmed. Section III explains the design of the proposed coherence protocol and Section IV presents its evaluation. Section V comments some related work and finally Section VI remarks the main conclusions of this work.

II. Background and Motivation

The aim of the hybrid memory model is to take advantage of the benefits of traditional cache hierarchies and local memories (LMs) putting them side by side. This section first explains the architecture of the hybrid memory model and how it is programmed to finally show the coherence problem it exposes.

A. Baseline Architecture

The hybrid memory model consists of extending a general purpose core with a LM a very simple programmable DMA controller (DMAC), as shown in Figure 1a.

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 load and store instructions to these virtual addresses. In order to distinguish whether a memory instruction has to be served by the cache hierarchy or by the LM a range check is performed on the virtual address prior to any MMU [13] action. If the virtual address is in the range reserved for the LM the MMU is bypassed and a physical memory address that points to the LM is generated [8]. This scheme has two important benefits. First, the access time to the LM is constant because no pagination is needed. Second, it allows to introduce 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 traditional cache hierarchy at all.

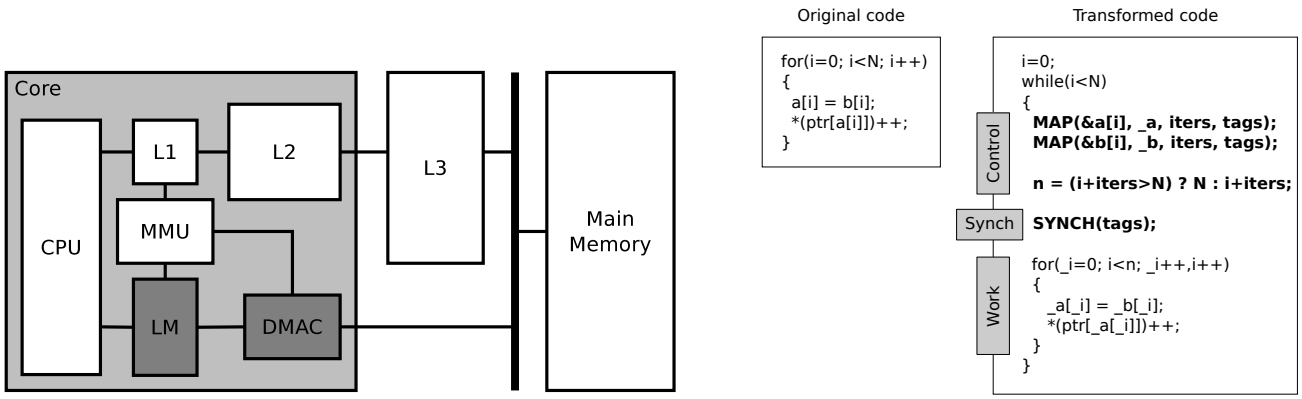
The DMAC is in charge of performing user-directed asynchronous memory transfers between the LM and the system memory (SM, which includes caches and main memory). It offers three basic operations: (1) *dma-get* transfers data from the SM to the LM, (2) *dma-put* transfers data from the LM to the SM and (3) *dma-synch* waits for the completion of certain DMA transfer operations. All three operations are explicitly triggered by software using store instructions to non-cacheable memory-mapped registers in the DMAC. The *dma-get* and *dma-put* operations are coherent with the SM. The L1 and L2 caches are write-through and all the levels of the hierarchy are inclusive. The bus requests generated by a *dma-get* snoop the L3 cache. If the data is present in the L3 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, in case the cache line is present in the L3 cache, it is invalidated. This mechanism guarantees memory coherence at the DMA transfer level [14]–[16].

B. Programming Model

In general, the introduction of a LM induces a particular execution model related to how the working set of the computation is mapped to the LM. Typically, the working set has to be broken in blocks, as the total amount of data typically exceeds the size of the LM, and these blocks have to be 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 1b. Each iteration of the outermost loop is broken down into 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 actual computation for the current block is performed. This code transformations are usually done by run-time libraries [9], [17], [18] or compiler transformations [19], [20].

Regular accesses are those memory accesses that expose very predictable access patterns (e.g., with a constant stride). These are mapped to the LM. Unpredictable memory accesses impose hard difficulties to be mapped to the LM [9], so they are served by the cache hierarchy. These are called *irregular accesses*. In Figure 1b, accesses to `a` and `b` are regular accesses, and the access to `ptr` is irregular.

In the control phase, for the regular accesses the data needed in the next work phase is brought to the LM (`MAP` primitive in Figure 1b). Blocks of data are copied from the SM to the LM, potentially sending back to the SM some previously used blocks of data. Notice that, 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 has to be done because otherwise, if only part of the block was modified, the write-back to SM would update the non-modified parts of the copy of the block 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 allocates them in fixed addresses. In Figure 1b 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 of the storage.



(a) Architecture. (b) Code transformation.

Figure 1: Architecture and operational model of the hybrid memory model.

The code in the work phase is very similar to the original loop, but with two differences. First, the code is transformed so that every instance of the innermost loop 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 memory 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 model appears when two incoherent copies of the same data (one in the LM and one in the SM) can be accessed along the computation. In particular, the problem arises because the compiler creates a copy of the data when maps it to the LM and, for regular accesses, it explicitly 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 two copies are incoherent, a modification in one path will not be noticed by the other path, so the execution can be incorrect.

Current compiler based solutions for this situation are very inefficient. Any approach relies on precise memory aliasing and data flow analysis [10]–[12]. In Figure 1b, this would translate in exactly predicting when, if ever, one instance of the `ptr` memory access aliases with any instance of the `_a` and `_b` accesses. In general, this problem is not yet solved, and a compiler would then adopt very 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 [9]. Doing so adds a big overhead because transfers of such small sizes are very inefficient. Software caching techniques are another option [9], [21]. These operate the LM with searches on software directory-like structures, keeping track of the LM content. Every potentially incoherent access in the computation is preceded by a lookup that diverts the access to the LM or to the cache hierarchy, causing an important loss of performance.

This paper proposes a lightweight and efficient mechanism that ensures coherency in a hybrid memory model. The solution skips 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 design of the coherence protocol for the hybrid memory model is based on a strict relation between hardware and software aspects. The main idea is to avoid maintaining two coherent copies of the data but, instead, ensure that the valid copy of the data is the one that is always used. 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, switching to a system state where only the valid version of the data exists, and then the eviction is performed. 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 resident in the LM. The DMAC updates the directory entries when it executes *dma-get* commands. On the software side, potential incoherent memory accesses are identified by the compiler and *guarded memory instructions* are emitted for them. The execution of a guarded memory

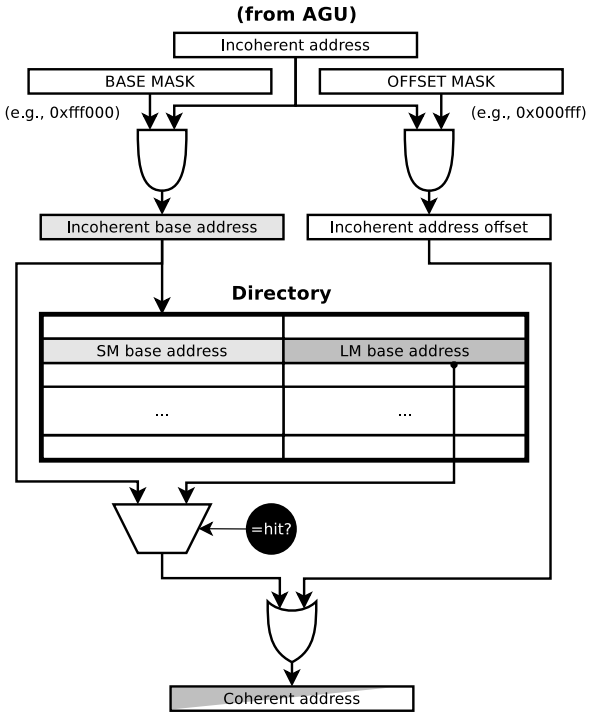


Figure 2: Directory usage in the address generation.

instruction will trigger a lookup in the directory that will divert the access to the memory that keeps the valid copy.

This design of the coherence protocol has several important advantages. First, the task of the compiler becomes trivial when it generates code for the hybrid memory model. Second, the coherence protocol can be implemented with simple hardware additions and simple compiler analysis. Third, the design does not introduce bus traffic nor overhead in codes that do not present coherence problems.

A. Hardware Design

The only hardware support needed for the implementation of the coherence protocol is a directory that keeps track of the data that is mapped to the LM. At the beginning of each loop the compiler statically partitions the LM into equally sized buffers. A directory entry is assigned to each of these LM buffers to map the starting address of the copy of the chunk of data in SM (i.e., the directory tag) to the starting address of the LM buffer where the chunk of data is mapped to. Then, the directory translates the addresses of a guarded memory instruction to a LM address if the original address aliases with any SM address in the directory (i.e., if some directory tag matches the address of the guarded memory access).

Note that as 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. In addition, the compiler inserts a loop prologue where it informs the hardware the LM buffer size. This is notified through a memory-mapped control register in the DMAC, which sets the values for the *BASE MASK* and *OFFSET MASK* internal registers. These two registers let the directory decompose any address into a base address and an address offset, that will be used to operate the directory and to translate addresses for any buffer size.

The *dma-get* operations update the directory contents. The destination LM address in the DMA command is used to identify the LM base address for that buffer. The source SM address of the DMA command is used to set the tag of the corresponding directory entry.

Figure 2 shows how the directory is used to change the address of a guarded memory access. The Address Generation Unit (AGU) [13] generates a potentially incoherent SM address (*Incoherent address*) according to the input operands of the guarded instruction. Notice that this is a SM address because it is generated by an irregular access. Two bit-wise AND operations between the *Incoherent address* and the *BASE MASK* and *OFFSET MASK* registers are performed to split the address in two, generating an *Incoherent base address* and an *Incoherent address offset*. The *Incoherent base address* is used to perform a lookup in the directory. In case of hit the memory instruction is accessing data in SM that is mapped to the LM, so the generated address in SM has to be replaced by the address of the copy in the LM. The base address of the corresponding LM buffer is retrieved from the directory (*LM base address*) and a bit-wise OR with the *Incoherent address offset* is applied, resulting in the *Coherent address*. In case the lookup misses there is no copy of the data in the LM, so the original SM address is preserved by performing a bit-wise OR between the *Incoherent base address* and the *Incoherent address offset*.

The implementation of the directory is composed of a Content Addressable Memory (CAM) for the *SM base address* and a RAM for the *LM base address*. The directory is restricted to have 32 entries to keep the access time low. According to CACTI [22], with a process technology of 45nm, the latency of the directory is 0.348 nanoseconds, so 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 not common. If a loop needing more than 32 buffers was found, the compiler could simply not map the exceeding regular accesses to the LM.

In order to support double buffering the directory contains an extra bit that indicates if the data of a LM buffer is being transferred into the LM by a *dma-get*. If a guarded memory access hits in the directory and this bit is set, the processor simply generates an internal exception until the bit is reset, at the *dma-get* completion. This is the simplest way to implement a safeguard against the case where a

guarded memory access accesses data that is currently being transferred into the LM using double buffer.

B. Compiler Support and Architecture-Specific Code Generation

This section outlines the main requirements on the compiler side to make an effective utilization of the hybrid memory model with the proposed coherence protocol.

One of the main benefits of the coherence protocol is that the algorithm that does the code transformation of a loop shown in Figure 1b becomes straightforward and is always safe, even if it encounters aliasing problems. 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. In order to do so first all the memory references in the loop are classified in regular and irregular accesses. For regular accesses it is easy to generate code that efficiently uses the LM [9], [20]. For irregular accesses memory aliasing analyses have to be applied. If the compiler can ensure an irregular access will never access data that is mapped to the LM it generates a memory instruction that will be served by the SM. If the compiler is not sure there can be aliasing the irregular access is a potentially incoherent access. Without a coherence protocol, when a potentially incoherent access is found, the compiler has to conservatively skip the code transformation. With the proposed coherence protocol, when a potentially incoherent access is found, the compiler emits a guarded memory instruction that the hardware will divert to the valid copy of the data.

One special case has to be treated separately. When the compiler does the aliasing analysis on an irregular reference and determines that there may be aliasing with some data mapped to the LM it generates a guarded memory instruction for that reference. In case this potentially incoherent access is for writing and the aliasing is with a chunk of data that is mapped for reading only the guarded memory operation 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 chunk of data to SM will be programmed and, when the buffer is reused to map new data, the *dma-get* operation will overwrite the modification done by the potentially incoherent store. The solution to this problem is to make the modification in the two memories. A simple way to do it is that the compiler generates a *double store*: one irregular store that will always 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 in case 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 both in performance and in power consumption. First, it has a very small performance impact because the second store is not dependent on the first one so they both can be issued in the same cycle. Second, the power consumption has a very small increase since the Load/Store Queue [13] will collapse the second store with the first one if it is not yet committed, having one single cache access.

The implementation of the guarded memory operations is highly dependant on the architecture. On a RISC architecture the instruction set should be extended to duplicate all load and store opcodes with a guarded form. As this might turn up to produce a high number of new opcodes, there are some other alternatives. One solution is to take unused bits that can be found on the binary representation of some instructions, as is the case of some opcodes in PowerPC. Another option is to provide a fewer range of guarded memory instructions and restrict the compiler to these. In the case of a CISC architecture like x86, where most instructions can access memory, instruction prefixes can be used to specify that the address of the memory operand has to be computed using the directory. Another option is to extend the ISA 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 itself, conceptually converting the guarded memory access to a coherency-aware address calculation plus a memory operation.

The approach taken in this paper targets a x86 architecture and uses an instruction prefix on any instruction that has one memory operand if the access to memory is potentially incoherent.

C. Data Coherency Management

This section proves the correctness of the coherence protocol. The two previous sections describe how memory operations are diverted to one memory or another when replication exists, considering that the valid copy of the data is in the LM. The next subsections prove that 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 a 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 3 describes the actions and states that can be applied to a piece of data. The *MM* state indicates the data is resident in main memory and has no replica neither in the cache hierarchy nor the LM. The *LM* state indicates that only one replica exists, and it is located in the LM. The *CM* state indicates that only one replica exists and it is located in the cache hierarchy. The

LM-CM state corresponds to a situation where two replicas of the data exist, one in the LM and the other in the cache hierarchy.

All actions identified with the "*LM-*" prefix 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 LM to SM. All 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 transition from the *MM* state to the *LM* state occurs when the software causes an *LM-map* action by executing a *dma-get*. Switching back to the *MM* state occurs when an *LM-unmap* action takes place 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, with an eventual cache line write-back in the latter case. 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 SM, and this ensures the system coherence as long as the data switches between the *LM* and *MM* states. Similarly, the cache coherence protocol ensures the system coherence as long as 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. If data is in the *LM* state because a previous *LM-map* occurred, a guarded memory instruction ($\texttt{ld}_{\texttt{guarded}}$ or $\texttt{st}_{\texttt{guarded}}$) will never cause a replica since the guard guarantees the access goes through the directory, and this will divert the access to the LM, so no action will take place in the cache hierarchy. Notice that the possibility of having unguarded memory instructions to SM ($\texttt{st}_{[\texttt{sm}]}$ or $\texttt{ld}_{[\texttt{sm}]}$) is discarded because the compiler never emits these instructions unless it is sure that there is no aliasing, so no replication can occur. When data is 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. Therefore, when two replicas are generated through a *LM→LM-CM* transition, it is ensured that both are identical. Along the execution the version in the LM will contain all the updates performed in the cache version plus all updates performed in the LM version using stores to the LM ($\texttt{st}_{[\texttt{lm}]}$). Therefore, the LM replica is the valid one to maintain coherence.

The *CM→LM-CM* transition happens due to an *LM-map* action. In that case, the DMA coherence ensures the two versions are identical by snooping the cache, as explained in Section II-A. Modifications using the double store update both versions, while $\texttt{st}_{\texttt{guarded}}$ and $\texttt{st}_{[\texttt{lm}]}$ modify the LM version only. Consequently, the two versions are identical or the LM one supersedes the one in the cache hierarchy.

In conclusion, it has been proven that 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 how the valid version is always selected at the moment of evicting a cache line to main memory.

2) Data Eviction: The transitions in the states diagram show 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 aspect of the hybrid memory model 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 cache version. 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 in the LM. Once the *LM* state is reached, at some point the program will execute a *dma-put* operation to write-back the data to the LM. 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 is in cache and it is valid, and the cache 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 the system first discards

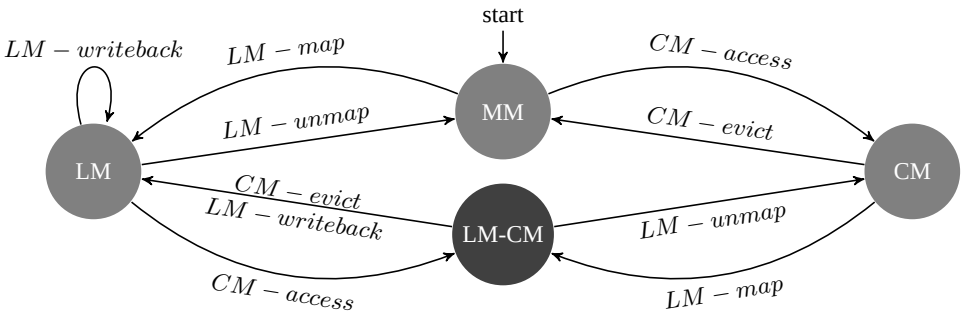


Figure 3: State diagram of possible data replication.

the invalid version and then, when only one valid replica exists, both the DMA and the cache coherence mechanisms correctly manage the eviction of the data to main memory.

IV. Evaluation

This section evaluates the coherent hybrid memory model. First it is evaluated the overhead of the coherence protocol. Then the performance of the coherent hybrid memory model is analyzed and compared with a traditional architecture exclusively based on a cache hierarchy.

A. Experimental Framework

The evaluation of the proposal has been done using PTLsim [23], extending it with a LM and the DMAC described in Section II-A. Table I shows the configuration parameters used for the simulations.

Five benchmarks from the NAS benchmark suite [24] have been selected for the evaluation. These benchmarks are typical HPC workloads: CG is a conjugate gradient algorithm, FT computes a Fourier transformation, IS does an integer sort, MG realizes 3-dimensional multigrid relaxation with periodic boundary conditions and SP solves multiple independent systems of non-diagonally dominant scalar pentadiagonal equations. Since the hybrid memory model is used only in the execution of computational loops the selected benchmarks have been studied on a per loop basis, 36 loops in total. Presenting per loop results allows to study the behavior of the proposal on different workloads and memory access patterns (regular and regular/irregular). Whole application results follow the same trends as the ones presented because they are vastly dominated by these computational loops. The 36 loops have been compiled using GCC 4.4.3 with the -O3 optimization flag on. 150 millions of x86 instructions have been simulated for each loop, except when the execution finishes before reaching that threshold.

B. Overhead of the Coherence Protocol

In order to study the overheads it has been designed a microbenchmark that stresses the coherence protocol. The microbenchmark consists of a loop that makes a computation using one single array, performing the operation `a[i+1]=a[i]+C`. The microbenchmark can be configured in 4 modes, as shown in Table II. The baseline mode represents the situation where the compiler is sure all memory references don't alias, so no guarded instructions are needed. The RD mode assumes the compiler does not know where the correct data for the reference `a[i]` is, so a guarded load is used to access memory. In the WR mode it is the write access to `a[i+1]` that is assumed to be potentially incoherent so a double store is generated, a first one that will go through the directory and a second one that will access SM. The RD/WR mode is a combination of the RD mode and the WR mode so guarded memory operations are used for the two references. In addition, the benchmark allows to adjust the percentage of memory operations that need to be guarded for every mode. Doing so it is possible to model loops with many references by applying loop unrolling to the loop body of the microbenchmark and then adjusting the number of memory references that need and don't need to be guarded. The aim of the microbenchmark is to facilitate the study of the overheads in several ways. First, it allows to study all possible scenarios in terms of number of memory accesses that are potentially incoherent. Second, it gives an upper bound of the overheads since the body of the loop is clearly dominated by the memory operations and the instruction-level parallelism is minimal because the code has a loop carried dependence and the majority of the instructions of every iteration also have dependences.

Figure 4 shows the overhead of different configurations of the microbenchmark both in execution time (Figure 4a) and in instructions committed (Figure 4b). In each figure three lines are plotted, 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 of the loop. The overheads are computed against the baseline mode of the microbenchmark, which has no potentially incoherent accesses.

Results of the RD mode show no overhead at all in instructions committed nor in execution time. The only additional code generated by the compiler when a potentially incoherent read access is found is a prefix in the load in-

Table I: PTLsim configuration parameters.

Parameter	Description
Issue scheme	Out-of-order. Fetch, Decode, Rename, Issue and Commit width of 4 instructions
Branch predictor	Hybrid 4K selector, 4K G-share, 4K Bimodal, 4K BTB 4-way and RAS 32 entries
Functional units	3 integer ALUs, 3 floating point ALUs and 2 load/store units
Register file	256 integer registers and 256 floating point registers
L1 I-cache	32 KB, 8-way set-associative, 64-byte lines, 2 cycles latency
L1 D-cache	Size depends on the configuration studied, 8-way set-associative, 64-byte lines, 2 cycles latency
L2 cache	256 KB, 24-way set-associative, 64-byte lines, 15 cycles latency
L3 cache	4 MB, 32-way set-associative, 64-byte lines, 40 cycles latency
Local memory	Size depends on the configuration studied, 2 cycles latency
Main memory	300 cycles latency

Table II: Scheme and configurable modes of the microbenchmark.

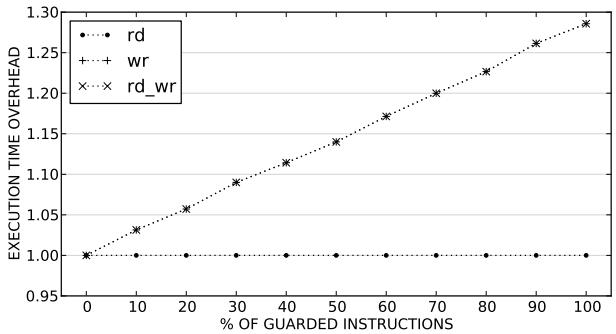
Microbenchmark	Mode	Description	Assembly code

`int a[SIZE];`
`int c;`
`for(i=0; i<N-1; i++) {`
`  a[i+1] = a[i] + c;`
`}`
	Baseline	No guarded instructions	`mov a(,esi,4),ebx`
			`add 0x0(c),ebx`
			`mov ebx,a+4(,esi,4)`

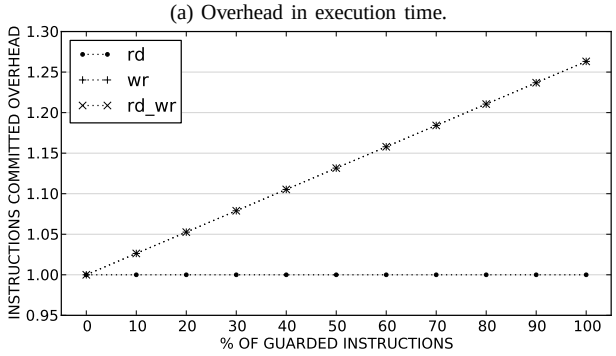
	RD	Guarded load for access a[i]	**`mov a(,esi,4),ebx`**
			`add 0x0(c),ebx`
			`mov ebx,a+4(,esi,4)`

	WR	Guarded store for access a[i+1]	`mov a(,esi,4),ebx`
			`add 0x0(c),ebx`
			`mov ebx,a+4(,esi,4)`
			`mov ebx,a+4(,esi,4)`

	RD/WR	Guarded load for access a[i]
		Guarded store for access a[i+1]
			`mov a(,esi,4),ebx`
			`add 0x0(c),ebx`
			`mov ebx,a+4(,esi,4)`
			`mov ebx,a+4(,esi,4)`



(a) Overhead in execution time.



(b) Overhead in instructions committed.

Figure 4: Overhead of the coherence protocol in all microbenchmark modes.

struction. This prefix will not be translated to any additional instruction at the decoding stage, so there is no overhead in committed instructions. At the microarchitecture level the only difference between a prefixed load and a normal one is that the former triggers a lookup in the directory in the execution stage, just after the AGU generates the address. This lookup fits in the cycle time so there is no overhead in execution time neither. In the WR mode it can be observed a linear overhead both in instructions committed and in execution time as the percentage of potentially incoherent accesses grow. Notice that the WR mode stresses the double store mechanism, which is not applied by the compiler if it can ensure the potentially incoherent write access aliases with a chunk of data that will be written back from the LM to SM. In the case only one guarded store would be generated the overhead would be zero, as in the case of a guarded load. The double store added by the compiler is translated to two instructions at the decode stage so the loop body grows from 7 to 9 instructions per iteration. This provokes an overhead of 26% in committed instructions when all write accesses are potentially incoherent, becoming only 13% when half of the write accesses are potentially incoherent (the double store is added in one iteration out of two) and goes down to less than 5.2% when 20% or less of the write accesses are potentially incoherent. These overheads in committed instructions have a direct relation with the overhead in execution time as it can be observed comparing Figures 4a and 4b. The double

store adds an overhead of 28% if it is used at every write access, and decreases to less than 10% when 35% or less of the write access are guarded and need the double store. The RD/WR mode of the microbenchmark is a sum of the two other modes. This means the situations described in the RD and in the WR mode happen together in this case, causing the overheads already shown. Since the RD mode has zero overhead, the RD/WR mode behaves exactly the same as the WR mode.

In conclusion, the coherence protocol adds no overhead when the potentially incoherent memory accesses are for reading data or when they are for writing and the compiler is sure that the double store is not needed. When the double store has to be added it provokes a maximum overhead of 28% in execution time and 26% in committed instructions. These are upper bound overheads. 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 that is performed in the loop has a much bigger weight that the one in the microbenchmark, so the expected overheads are far from the reported upper bounds.

Figure 5 shows the overhead of the coherence protocol in real benchmarks. The overhead is computed comparing the execution time of the benchmarks running on a hybrid memory model with a 32KB LM extended with the coherence protocol against a hybrid memory model with a 32KB LM without the coherence protocol. It has been checked that incorrect results in the baseline executions due to coherence problems don't change the behaviour of the loops. In 34 of the 36 loops the overhead 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 the FT-1 and IS-3, which present overheads of 3.35% and 3.03%. The overhead in both cases is due to the increase in instructions, as explained in the WR mode of the microbenchmark, though the overheads in these two loops are far from the upper bound shown. In the FT-1 this is because the loop does complex operations on floating point data, which has a very high cost compared to the one introduced by the double store. In the IS-3 the computation is very simple but the number of guarded memory accesses is 1 out of 7. On average, the overhead is negligible, less than 0.2%.

C. Performance Analysis

An immediate result of the proposal in this paper is that any computational loop can now be executed on the hybrid memory model with no restrictions coming from coherence problems. This section compares the performance of the coherent hybrid memory model with a purely cache-based architecture following 3 steps. Initially speedup numbers are reported. Then the number of instructions committed for both architectures is studied. Finally it is shown the impact of the hybrid memory model in stall cycles and cache misses. The following subsections correspond to each one of these steps.

1) Speedup: Speedup number for the loops can be observed in Figure 6. For each loop three stacked bars are plotted, one per LM size: 16KB, 32KB and 64KB. The speedup shown corresponds to the speedup of the execution of the loop on the hybrid memory model with a LM of a given size against the execution of that loop on a cache-based architecture. For fairness, when a hybrid memory model with a LM of a given size is compared against a cache-based architecture, the original capacity of 32KB of the L1 D-cache of the cache-based architecture is increased by the size of the LM. In addition, the bars are stacked in such a way that they show the weight of each phase of the execution in the hybrid memory model. All studied loops present some degree of improvement, except three of them that present slowdowns: CG-04, IS-1 and IS-2. These loops just initialize one vector to zero. Some loops present modest speedups, ranging between 1.05x and 1.20x. Other loops present noticeable speedups, ranging from 1.25x up to 2x. The loops CG-03, CG-08 and IS-04 present impressive speedups, ranging from 2.50x to 3.50x. These loops copy the contents of a vector on another one.

In conclusion, the average speedups are 1.47x, 1.49x and 1.50x with LMs of 16KB, 32KB and 64KB, respectively. The first observation to take into account is that the size of the LM does not affect much the performance. The reason for that corresponds to a key aspect of the LM utilization. There is a relation between the number of memory references mapped to the LM and the size of the buffers that are allocated in it. In general, the bigger the LM, the bigger the buffers and, therefore, the work phases are longer and less control phases have to be executed. A 16KB LM is enough to operate with buffers big enough so that the overheads coming from the control phase are very low, 2.6% on average. This result, although not being a direct consequence of the proposal in this paper, emphasises the importance of solving the programmability issues surrounding the utilization of a LM.

2) Instructions Committed: The introduction of a LM requires extra instructions to be executed to orchestrate the computation and data transfers. Figure 7a shows the overhead in instructions committed for the loops. The overhead is computed dividing the instructions committed in the executions on a hybrid memory model with a 32KB LM by the instructions committed of the executions on a cache-based architecture with 64KB of L1 D-cache. For LMs of 16KB and 64KB the results follow the same trends. Again, stacked bars to see the distribution between the three execution phases are plotted. The average increase in instructions committed for all loops is 9.77%. Most loops present a very modest increase on this metric, staying below 10%, while a few of them present close to a 25% overhead

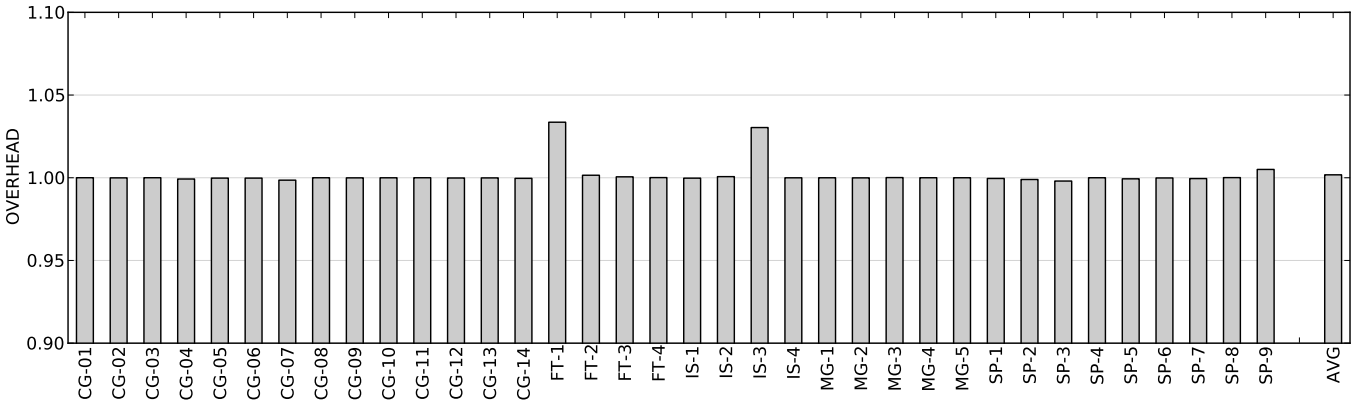


Figure 5: Overhead of the coherence protocol in real benchmarks.

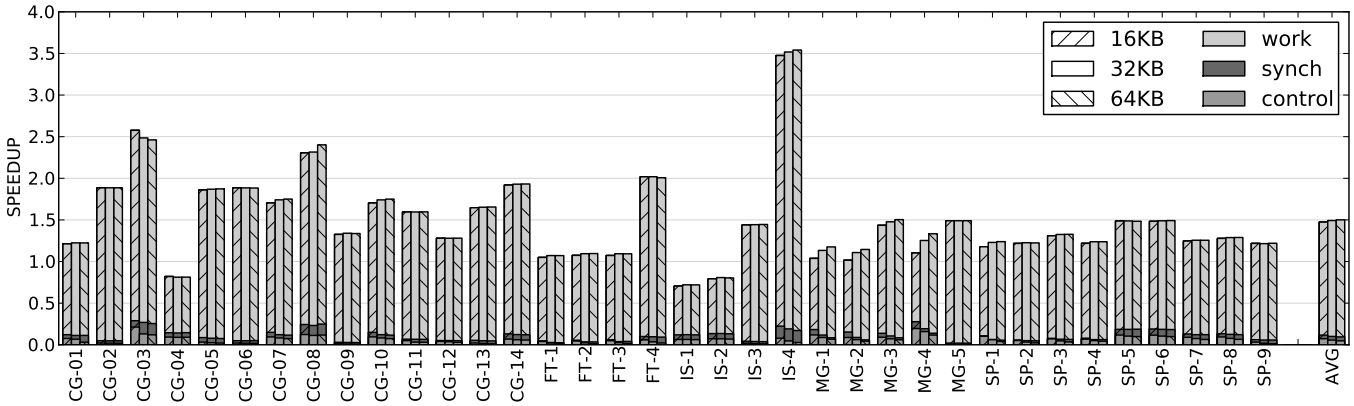
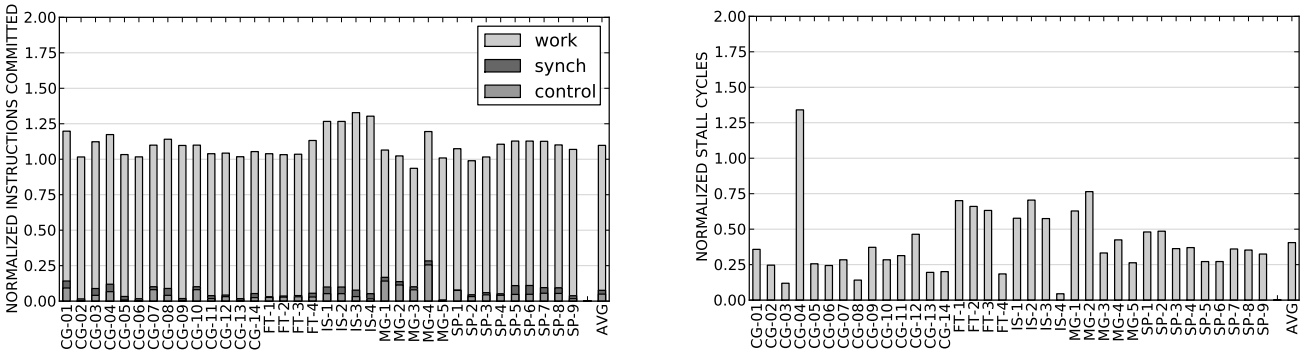


Figure 6: Speedup of the hybrid memory model against a cache-based architecture.

or even higher. The general trend is that the increase of instructions committed is related to the extra instructions added in the control and synchronization phases, as a result of having very similar number of instructions committed in the work phase of a hybrid memory model and in the whole cache-based execution.

In conclusion, it is important to observe that there is a not negligible increase in instructions committed when a loop is transformed to work with the hybrid memory model. This overhead in instructions committed is due to the additional phases of control and synchronization. In any case, in the previous subsection it has been shown that the overhead in cycles provoked by these extra instructions is affordable. In addition, paying the penalty of these two phases is worth it thanks to the benefits that the LM brings in the work phase, as it is addressed in the next subsection.

3) Stall Cycles and Cache Misses: This section explains the speedup numbers previously described by showing the reduction on stall cycles and on cache misses achieved by the hybrid memory model. Figure 7b shows the reduction in stall cycles. Stall cycles for the work phase of the execution of the loops on a hybrid memory model with a 32KB LM have been normalized against a cache-based architecture with 64KB of L1 D-cache. For LMs of 16KB and 64KB the results follow the same trends. It can be observed that stall cycles in the work phase are drastically lower with the hybrid memory model, an average factor of 0.40. There is a clear relation between the reduction in this metric and the achieved speedups. Loops exposing modest speedups also suffer from a modest reduction in stall cycles and loops with significant speedups have a much bigger reduction. The improvement is caused by the introduction of the LM, which serves the regular accesses and never introduces stall cycles. This doesn't happen in cache-based architectures because caches introduce stall cycles when there is a cache miss. Figure 8 shows this situation. In it the reduction of cache misses in all the levels of the cache hierarchy can be observed, comparing executions on the hybrid memory model with a 32KB LM against its cache-based counterpart. The average reduction in all levels is huge, achieving factors of 0.16 for L1, 0.08 for L2 and 0.03 for L3. Two situations explain these reductions. First, the number of accesses to the cache hierarchy is much lower in the hybrid model because of the introduction of the LM. Second, these reduced amount of memory accesses that are served by the cache hierarchy have a much higher hit rate because the hybrid memory



(a) Instructions committed.

(b) Stall cycles.

Figure 7: Effect of the hybrid memory model on instructions committed and stall cycles.

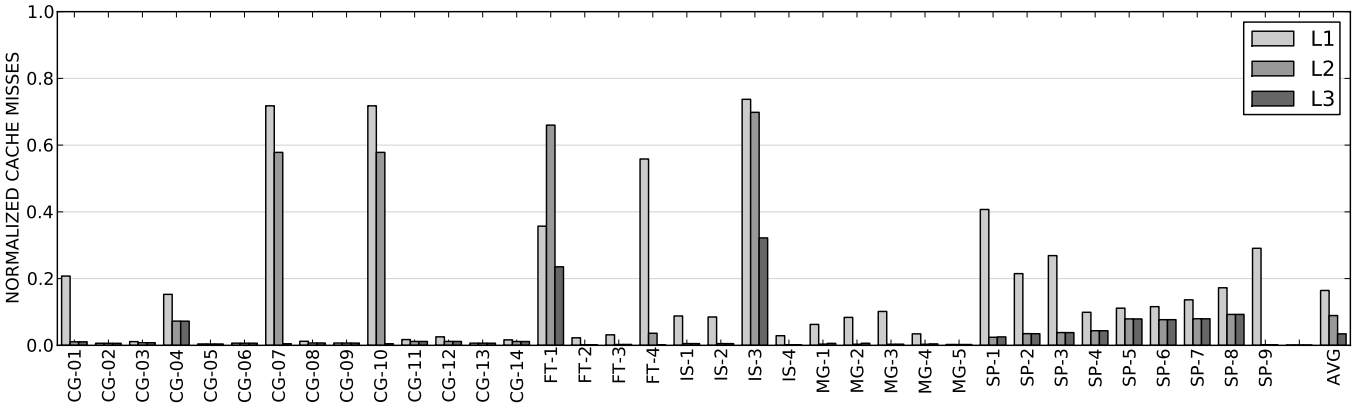


Figure 8: Effect of the hybrid memory model on cache misses.

model maps a lot of data to the LM, leaving the cache hierarchy for the structures of the software cache and the data accessed by irregular accesses, enormously reducing the number of misses due to conflicts.

In conclusion, the hybrid memory model succeeds in accelerating the computational loops by drastically reducing the stall cycles along the execution. This reduction is due to the introduction of the LM because it serves the majority of memory accesses that traverse big data sets without causing any stall. The LM also removes a lot of pressure on the cache hierarchy, which is now only used to serve a reduced amount of memory operations that operate on smaller data structures, so the number of misses decrease.

V. Related work

The idea of adding a LM side to the cache hierarchy is not novel. Bertran et al. [8] propose such organization in a general purpose core. This is also present in a commercial product like the NVIDIA Fermi GPGPU [7]. These two approaches do not solve the coherence problem between the two storages. Bertran et al. [8] give the compiler the responsibility to discard loop transformations in case of coherence problems, restricting the effective utilization of the hybrid memory model. In the NVIDIA Fermi [7] the global memory, which is cached, and the LM are also incoherent and the programmer is the one in charge of explicitly managing 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 that declares and manages them, since neither the hardware nor the compiler give any support for coherency management.

Cohesion [25] attacks the scalability problem of large CMPs using a different approach. Cohesion allows the software to dynamically select which cache lines are cache coherent. This is accomplished by enabling and disabling the cache coherence protocol for these lines. This approach faces the same problem as the hybrid memory model because it opens the door to incoherent copies of data. In Cohesion it is the programmer who explicitly manages the incoherent copies of data. In contrast, in the coherent hybrid memory model it is the proposed software/hardware mechanism that handles them.

This paper relies on previous works on DMA coherence. The IBM Cell architecture [6], [14], [15] ensures DMA coherence through snoop mechanisms in DMA transfers.

In this 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 made visible to other cores DMAs are used so the coherence is ensured. In the hybrid memory model this approach has to be extended to support coherence at the memory instruction level because a core can access both memories.

For IO devices, DMA coherence can be supported by software and hardware mechanisms [16]. The former rely on explicit flush operations on the cache hierarchy in order to ensure both the device and the CPU get the latest version of the data. Alternatively, it is also possible to avoid flushing operations by defining non-cacheable memory regions. Hardware solutions are based on a specific component, the coherence manager, which centralizes all the IO operations and channels data between the cores and the devices. The approach in this paper discards the flushing of data in the cache hierarchy. Non-cacheable regions are also discarded because of the inherent complexity of identifying which regions have to be cacheable and which are not. The proposal in this paper includes the coherence manager also, and extends it with the directory to keep coherence at the memory instruction level.

D. Tang et al. [26] 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 separated from the data caches creates potential incoherences that are solved by a refinement of the MOESI and ESI cache coherence protocols. In the coherent hybrid memory model 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 introduction of a hybrid memory model that combines a local memory and a cache hierarchy can have a great impact on multicore architectures by reducing its power footprint and achieving a better scalability. Local memories are known to be much more power-efficient than caches and, in the context of shared memory architectures, they can significantly reduce the coherence traffic associated to the sharing of the cache hierarchy. Unfortunately, the introduction of a local memory opens the door to coherence problems between the two storages and also imposes important programmability difficulties associated to the orchestration of data transfers.

The main contribution of this paper is the design of a coherence protocol for hybrid memory models. The design admits that data can be replicated in the two storages, but avoids a solution based on the interconnection of both storages and the generation of any coherence traffic. Instead, the design is based on a very simple hardware addition consisting of a directory that keeps track of the contents of the local memory and a simple software mechanism that identifies potentially incoherent memory accesses. The address calculation of the potentially incoherent memory accesses goes through the directory, effectively diverting the access to the memory that keeps the valid copy of the data. Code transformations needed to operate the resulting architecture can be done with a simple compiler algorithm that does not need to care about complex memory aliasing analysis. As a result, the task of the compiler becomes the identification of suitable memory references to map data to the local memory, plus the introduction of guarded memory instructions for memory accesses that potentially alias data in the local memory.

The evaluation of the proposal shows that the overhead of the coherence protocol is negligible and that the benefits of a hybrid memory model are significant in two aspects. It achieves an average speedup of 1.5x among a representative set of numerical HPC computational loops taken from the NAS benchmark suite. In addition, the cache hierarchy utilization is much lower and the number of misses in all levels are drastically reduced.

In conclusion, the proposed coherence protocol defines a promising alternative for the definition of the memory subsystem organization of future multicore architectures.

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*, 2007.

[2] M. Kandemir, O. Ozturk, and M. Karakoy, "Dynamic On-Chip Memory Management for Chip Multiprocessors," in *CASES '04*. ACM, 2004, pp. 14–23.

[3] R. Murphy, "On the Effects of Memory Latency and Bandwidth on Supercomputer Application Performance," in *IISWC '07*. IEEE Computer Society, 2007, pp. 35–43.

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

[5] 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*. ACM, 2002, pp. 73–78.

[6] J. Kahle, "The Cell Processor Architecture," in *MICRO 38*. IEEE Computer Society, 2005, pp. 3–4.

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

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

[9] 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*. ACM, 2008, pp. 292–302.

[10] W. Landi and B. G. Ryder, "A Safe Approximate Algorithm for Interprocedural Aliasing," in *PLDI '92*. ACM, 1992, pp. 473–489.

[11] A. Deutsch, "Interprocedural May-Alias Analysis for Pointers: Beyond k-limiting," in *PLDI '94*. ACM, 1994, pp. 230–241.

[12] R. P. Wilson and M. S. Lam, "Efficient Context-Sensitive Pointer Analysis for C Programs," in *PLDI '95*. ACM, 1995, pp. 1–12.

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

[14] M. Gschwind, "Optimizing Data Sharing and Address Translation for the Cell BE Heterogeneous Chip Multiprocessor," in *ICCD '08*. IEEE Computer Society, 2008, pp. 478–485.

[15] M. Kistler, M. Perrone, and F. Petrini, "Cell Multiprocessor Communication Network: Built for Speed," *IEEE Micro*, 2006.

[16] T. B. Berg, "Maintaining I/O Data Coherence in Embedded Multicore Systems," *IEEE Micro*, 2009.

[17] J. Balart, M. Gonzalez, X. Martorell, E. Ayguade, Z. Sura, T. Chen, T. Zhang, K. O'Brien, and K. O'Brien, "A Novel Asynchronous Software Cache Implementation for the Cell-BE Processor," in *LCPC '07*. Springer-Verlag, 2007, pp. 125–140.

[18] S. Seo, J. Lee, and Z. Sura, "Design and Implementation of Software-Managed Caches for Multicores with Local Memory," in *HPCA '09*. 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™ Architecture," 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*. IEEE Computer Society, 2005, pp. 161–172.

[21] T. Chen, T. Zhang, Z. Sura, and M. G. Tallada, "Prefetching Irregular References for Software Cache on Cell," in *CGO '08*. ACM, 2008, pp. 155–164.

[22] P. Shivakumar, et al. CACTI 3.0: An Integrated Cache Timing, Power, and Area Model. 2001.

[23] M. T. Yourst, "PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator," in *ISPASS '07*. IEEE Computer Society, 2007, pp. 23–34.

[24] 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*. IEEE Computer Society, 1991, pp. 158–165.

[25] 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*, 2011.

[26] 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*. IEEE Computer Society, 2010, pp. 1–12.