# Software Caching Techniques and Hardware Optimizations for On-Chip Local Memories

Nikola Vujić

Advisors: Marc Gonzàlez and Eduard Ayguadé

Department of Computer Architecture
Universitat Politècnica de Catalunya

A thesis submitted for the degree of
*Doctor of Philosophy*

March 2012

To my sister and parents

# Acknowledgements

To my advisors Marc Gonzàlez and Eduard Ayguadé for their patient guidance. Without them this thesis would not be possible.

To Xavier Martorell, Alex Ramírez, Felipe Cabarcas and Lluc Álvarez for their significant contribution to this thesis.

To Roger Ferrer, Alex Duran, Paul Carpenter, Pieter Bellens, Javier Teruel, Nikola Puzović, Milan Pavlović, Bojan Marić and other colleagues from Barcelona Supercomputing Center for their collaboration, help, and insightful reviews.

To my family and friends for their support, providing it when it was most needed.

At the end, I would like to acknowledge the support of the European Commission through the HiPEAC-2 Network of Excellence (FP7/IST 217068), the support of the Spanish Ministry of Education (TIN2007-60625, and CSD2007-00050), the Generalitat de Catalunya (2009-SGR-980), FPI scholarship (BES-2008-008885) and the BSC-IBM MareIncognito project.

# Abstract

Despite the fact that the most viable L1 memories in processors are caches, on-chip local memories have been a great topic of consideration lately. Local memories are an interesting design option due to their many benefits: less area occupancy, reduced energy consumption and fast and constant access time. These benefits are especially interesting for the design of modern multicore processors since power and latency are important assets in computer architecture today. Also, local memories do not generate coherency traffic which is important for the scalability of the multicore systems.

Unfortunately, local memories have not been well accepted in modern processors yet, mainly due to their poor programmability. Systems with on-chip local memories do not have hardware support for transparent data transfers between local and global memories, and thus ease of programming is one of the main impediments for the broad acceptance of those systems. This thesis addresses software and hardware optimizations regarding the programmability, and the usage of the on-chip local memories in the context of both single-core and multicore systems.

Software optimizations are related to the software caching techniques. Software cache is a robust approach to provide the user with a transparent view of the memory architecture; but this software approach can suffer from poor performance. In this thesis, we start optimizing traditional software cache by proposing a hierarchical, hybrid software-cache architecture. Afterwards, we develop few optimizations in order to speedup our hybrid software cache as much as possible. As the result of the software optimizations we obtain that our hybrid software cache performs from 4 to 10 times faster than traditional software cache on a set of NAS parallel benchmarks.

We do not stop with software caching. We cover some other aspects of the architectures with on-chip local memories, such as the quality of the generated code and its correspondence with the quality of the buffer management in local memories, in order to improve performance of these architectures. Therefore, we run our research till we reach the limit in software and start proposing optimizations on the hardware level. Two hardware proposals are presented in this thesis. One is about relaxing alignment constraints imposed in the architectures with on-chip local memories and the other proposal is about accelerating the management of local memories by providing hardware support for the majority of actions performed in our software cache.

# Contents

# List of Figures

# Chapter 1

# Introduction

This chapter is the introductory chapter of this thesis. It is organized in five sections, as follows. In Section 1.1, we start with background on the evolution of the microprocessors from single cores to muti-cores. In Section 1.2, we discuss some of the challenges for the design of the multi-core processors. Introduction of on-chip local memories appears in Section 1.3. This section introduces some of the main architectural considerations for the inclusion of on-chip local memories in processors, and it presents the major programmability issues of the architectures with on-chip local memories. Finally, we switch to describing the contributions and the structure of the thesis. Contributions are described in Section 1.5, and the structure of the rest of the document is described in Section 1.6

## 1.1 From a Single Core to a Multi-Core

During the last decade, the computer processors industry has taken a particular path towards parallel computing and including multiple cores on a chip. Chip Multiprocessors (CMPs) are key aspect of computer architecture today and they can be found in almost every computer system. There are many reasons why multi-cores appeared on the market and completely changed the previous single-core era.

During past two decades (in the 1980s and 1990s), vendors focused on designing single-core CPUs that would run single threaded applications. All that time, the performance increase from generation to generation was easy to see. It was achieved mostly by an increase in frequency. Frequency was the most important source of delivering better computer performance at that time. In Figure 1.1 [1], we can see that frequency has been scaling for 30 years (from 1971 to 2000) and then in the last 10 years almost flat line is observed. Officially, the end of the frequency scaling was announced in May 2004 by Intel. Intel was working on the Tejas processor that was to be a successor of Pentium 4 processor. The Tejas processor run at very high frequency, around 7GHz, but that high clock frequency caused power consumption and heat dissipation at detrimental levels which led Intel to cancel the project. In Figure 1.2 [2], we can see that the power density reached relatively high level even

[1] Manually drawn figure using online sources (e.g., wikipedia and processors technical documentation).

[2] Manually drawn figure using online sources (e.g., wikipedia and processors technical documentation).

in 2000 with Pentium 4. It was not too far from the power density of nuclear reactor, while the power density of heating plate was surpassed more than 15 years ago (see Figure 1.2). Therefore, it became mandatory not to increase power density anymore and then frequency stopped scaling due to power restrictions. This led engineers to find a new way to deliver

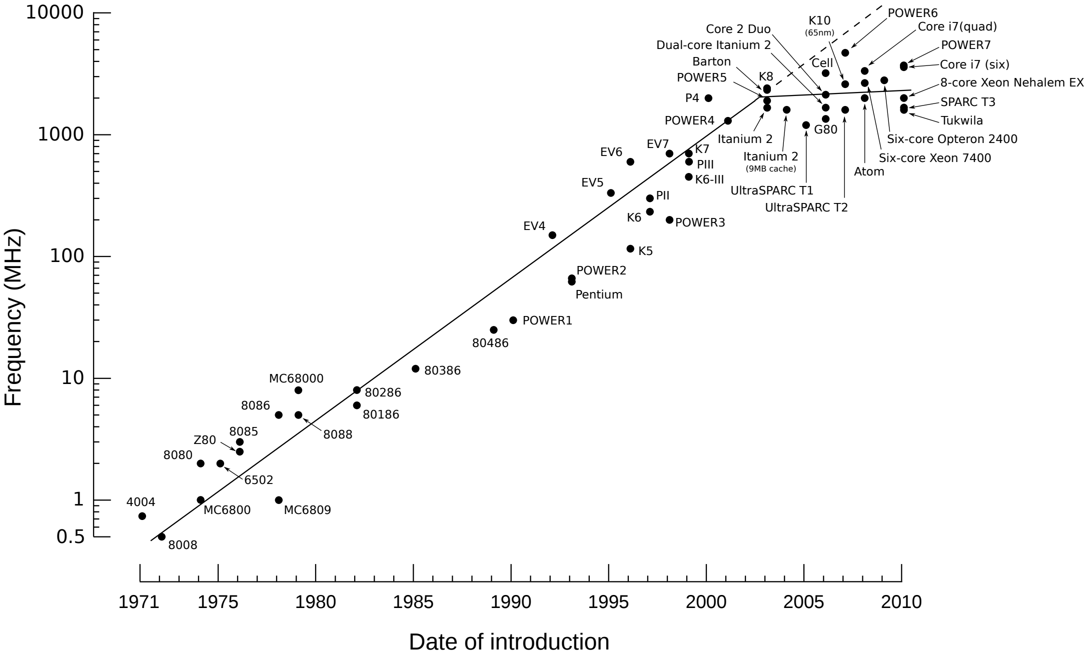


Figure 1.1: Frequency scaling.

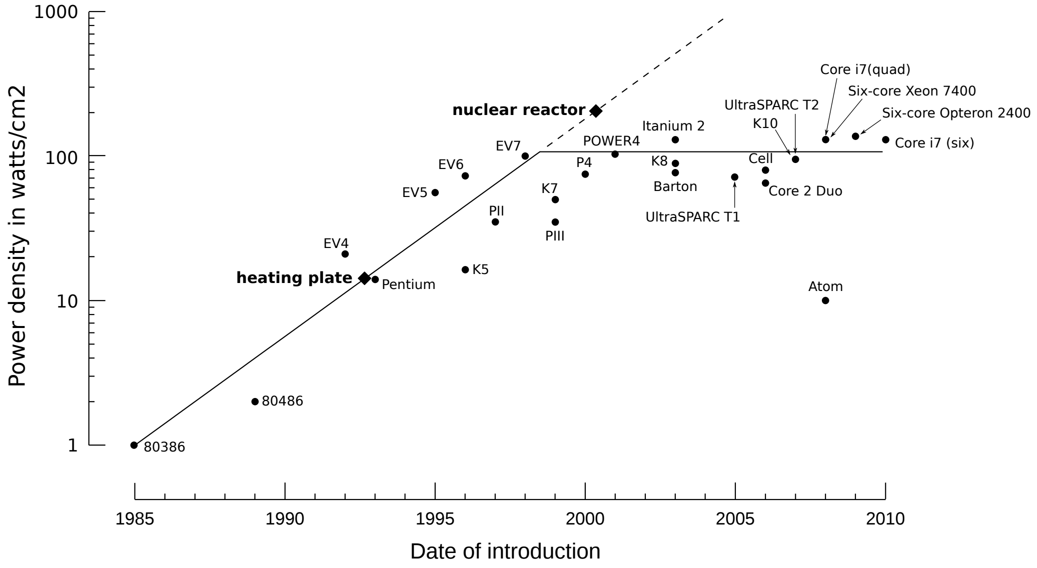


Figure 1.2: Power density.

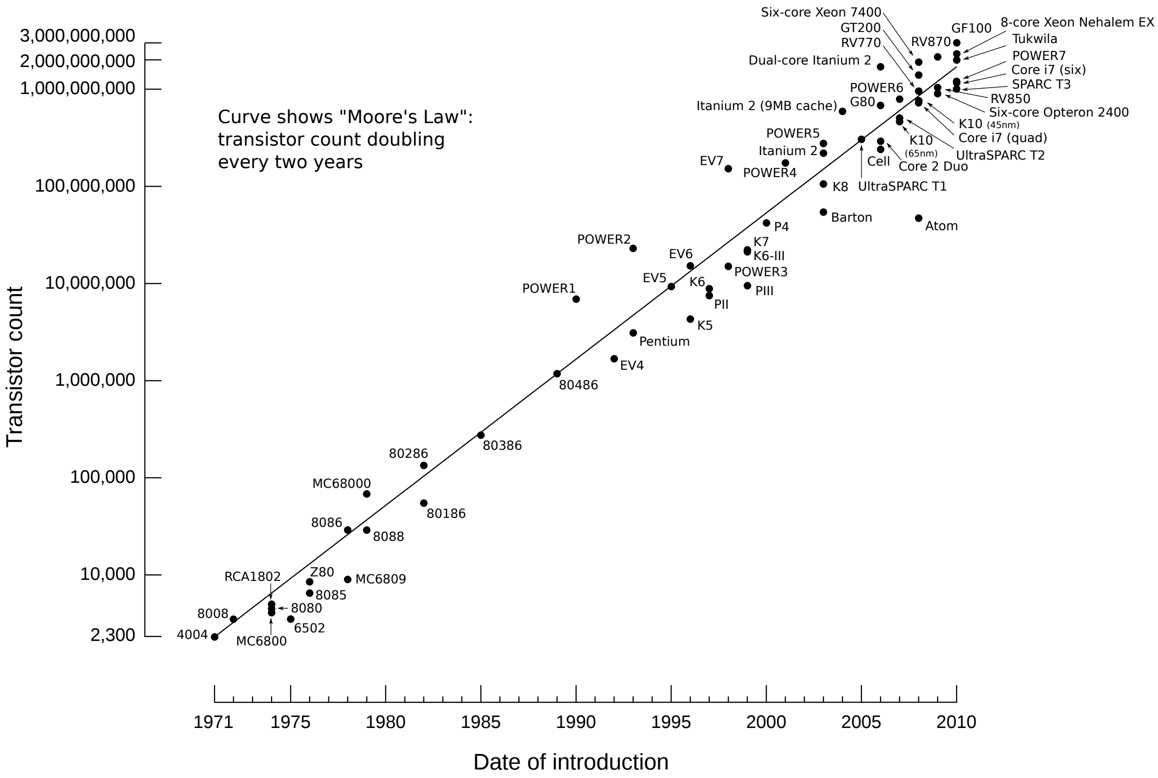


Figure 1.3: Moore's law.

better performance coming from sources other than the increased clock frequency.

The only available resource to be used was the increased transistor count. In Figure 1.3 [1], we have plotted the transistor count for a variety of microprocessors, dating from 1971 till 2010. All of them follow the same trend, the transistor count has been doubling every two years. This trend is known as Moore's Law [73, 74] and it has been guiding computer architecture for four decades so far and it is still valid today. From year to year, more space has been available on a chip for new hardware. So, how to use this additional space has become a big matter. Surely, one option could have been to aggressively improve core microarchitecture (out of order execution, speculative execution, branch prediction, larger caches etc) but this strategy reached a point of diminishing returns - a huge number of transistors was devoted to hardware that did not give enough performance improvement in return (performance per transistor started to decrease). At this point engineers started to think about putting multiple cores on a single die [80]. This became a reality in 2001 when industry began to introduce Chip Multirocessors. The first CMP was IBM POWER4 [108, 116] processor with two cores on a die, debated in 2001. Today, there is no high performance processor family without a multi-core design.

Multiple cores on a single die improve computer performance since two or more cores can share the same workload and, by doing so, application throughput increases. But the

[1] Manually drawn figure using online sources (e.g., wikipedia and processors technical documentation).

performance improvement is not the only reason for multi-core design, the power density has become as important as performance is. With a multi-core design it is possible to gain performance and to keep the power density within a reliable range. In Figure 1.2, we can see that in the last decade (decade of multi-cores) the power density has not scaled but before it has been doubling every two years. Today, all multi-cores keep power density around 100 $watts/cm^2$ (see Figure 1.2) which is very near to the power density of some single-cores such as the Intel Pentium 4 processor.

## 1.2 Challenges in Multi-Core Era

Multi-core era has brought many hardware and software challenges. It is expected to have many-core CMPs (e.g. 128 cores or more) in the near future but nobody knows how to efficiently deal with that number of cores for real and non trivially parallel applications, how to program it and how to build good hardware. Along with the increased number of cores, scalability becomes an immediate issue. Putting more cores on a chip is meaningless if the computer performance does not scale when more hardware resources are present on a die. The scalability issue is mostly related to the memory system organization since multiple cores must communicate in order to synchronize when they work on the same workload and this communication is supported by the memory system. In a case of intensive communication needs, memory system becomes a bottleneck for good scalability. It is not yet clear what memory organization is the best, whether to put on-chip coherent caches, non-coherent caches, or even local memories. Additionally, power efficiency, as one of the primary metrics for processors designs in the multi-core era, imposes a big challenge for the selection of the proper memory system for building future power-aware CMPs. It can be said that the power density of the existing multi-cores is on the edge and it shouldn't go higher. It is not clear how to build many-core CMPs and to keep power density, at least, at the level of the existing processors.

All mentioned issues and challenges have stimulated many design studies for CMPs to be present in research and development today. For instance, the majority of existing general purpose CMPs are Shared Memory MultiProcessors (SMPs). The appearance of the shared memory CMPs has been dictated mainly by the software community since through the history of parallel computing, shared memory models have been well accepted by programmers due to transparent view of shared memory space. So, programmability has dictated the memory system organization for the majority of existing CMPs. On example, IBM Power7 [115] includes 8 cores with 256KB L2 private and coherent caches plus a big shared L3 cache of 32MB. Sun SPARC T3 [101] has 16 cores and each core has L2 coherent cache of 6MB but no shared cache level. AMD Opteron [2] has 16 cores with 1MB coherent L2 caches and 16MB shared L3 cache. Intel Xeon E7 [3] has ten cores with 256KB coherent L2 caches and one shared L3 cache of 30MB. It is common for majority of existing SMPs to have relatively small number of cores (up to 16) but to devote a big number of transistors to the cache hierarchy and to the hardware support for cache coherence. All of them scales well due to a relatively small number of cores but their cache coherence protocols are not projected to scale well for many-core SMPs. Both the power consumption originated in the memory hierarchy and the scalability of the memory coherence protocols constrain the sharing and the size of caches when cores are replicated up to a certain level [57, 62, 68, 78, 94, 114]. As an immediate consequence, the memory hierarchy is expected to evolve up to some novel organization that satisfies the scalability requirements and avoids potential bottlenecks in the shared levels of the memory hierarchy.

One of the solutions for large multi-core architectures (many-core CMPs) is to take a form of a distributed memory system on a chip. The recent Intel Single-chip Cloud Computer (SSC) architecture [69] is a clear example of this type of solution. It includes 48 simple cores organized in 24 homogeneous tiles. All cache levels are private to each core (16KB L1 data cache and 256KB L2) without any cache coherence protocols implemented. Besides to this reduced amount of cache memory there is a 16KB local memory to accelerate message-passing operations, shared between the two cores in every tile. This approach forces programmers to use distributed memory programming paradigms such as Message Passing Interface (MPI) which is well accepted in research but not in the commercial development. This kind of memory system organization is good example for how much programmability can be affected by a different memory system organization.

Another possible solution to the lack of power efficiency and scalability of current coherence protocols for SMPs is the introduction of on-chip local memories, also known as scratchpad memories [13].

## 1.3 On-chip Local Memories

The key element of the study in this thesis is on-chip local memory. We study its introduction, programmability, and optimizations regarding its usage in the context of single-core and multi-core systems.

Local memories have been successfully introduced in heterogeneous CMPs like the Cell BE [52, 56]. In addition, recently they have been introduced side to the cache hierarchy in GPGPUS [41] and in general-purpose cores [15] to form what is called a hybrid memory model. The advantages of local memories are that they offer similar performance as caches do but in a much more power-efficient way [13], and moreover they offer fast and constant access time without generating any coherence traffic. Also, local memories can feed functional units with relatively big chunks of data (bigger than cache lines) without producing stall cycles as frequently as caches do due to cache miss penalties. This property is especially desirable for Single Instruction Multiple Data (SIMD) functional units. Data transfers to/from local memories are supported by Direct Memory Access (DMA) engines but they are under software control. Data transfers are explicitly controlled by programmers and can be overlapped with computation which enables programmers to address another important limitation of conventional architectures, known as the memory wall.

### 1.3.1 Architectural considerations

In Figure 1.4, we present high level design of a processor with local memory. We can see in Figure 1.4 that three main blocks appear: CPU, Local Memory, and DMA engine. Local

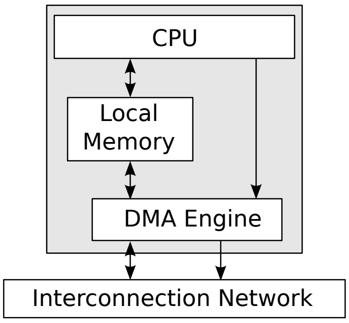


Figure 1.4: Local Memory and DMA engine.

memory is used to feed functional units in the CPU with the data while the DMA engine is used to move the data between the local memory and the main memory. It is important to emphasize that if we need data from the global memory, then we have to explicitly program the DMA engine to transfer it. So, all data transfers to or from the local memory are under software control. Programming data transfers consists of a simple communication to the DMA controller of a DMA engine in order to schedule a DMA command for the transfer. Three basic DMA commands are: *dma-put*, *dma-get*, and *dma-synch*. The *dma-get* command transfers data from the global memory to the local memory while the *dma-put* does it in the opposite direction. The main parameters of these two commands are: source address, destination address, size of the transfer and a tag to be assigned to the transfer. The *dma-synch* is used to synchronize with the data transfers associated to a particular set of tags.

In the context of the cache coherency it is important to mention what is happening when local memories are mixed with caches. In Figure 1.5, we show a hybrid multi-core system with three type of cores with different memory organization that may exist when local memories are taken into consideration.

- The first core on the left is the traditional core without local memory and only with the cache hierarchy. This type of core is the conventional general purpose core like the most cores in the existing SMPs. In the hybrid CMPs this type of cores are used to run operating system (e.g. PPE in Cell BE).

- The core in the middle is the core with the local memory only. This type of core is not general purpose core since cache hierarchy is missing and then it is not convenient to process general purpose workloads in this type of core. This kind of cores are accelerator cores (very optimized to accelerate computation intensive workloads) in the existing systems (SPE in Cell BE, GPGPU).

- The last core on the right is the core with the hybrid memory model where local memories exist aside a cache hierarchy in the same core (e.g. Intel's SCC cores have similar organization). In this type of cores local memories can exist independently of caches, being completely managed by software, or local memories can have tag entries

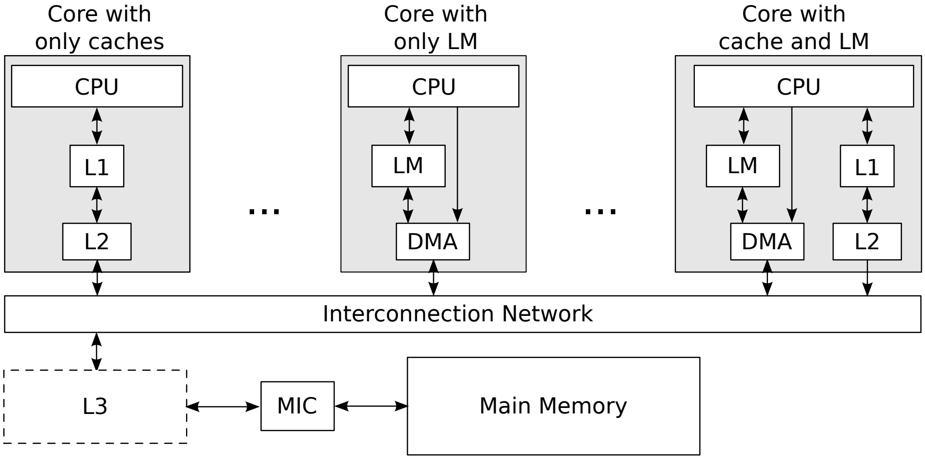


Figure 1.5: Memory system variations with and without local memories.

into a L1 cache, making it a software managed cache where the cache accesses get either data from the local memory or from the L1, depending on where the data is.

- Finally, all cores may have a shared L3 cache, meaning that all memory transfers (DMA transfers for local memories and read/write transfers for L2 caches) generated by the cores in the multi-core system go through a shared cache level that is responsible of communicating to the main memory through the Memory Interface Controller (MIC) which is directly connected to the shared cache level in this case. If a shared cache level does not exist then MIC is directly connected to the Interconnection Network.

Cache coherency in these systems is maintained on a DMA transfer level [43]. Local memories are not coherent but DMA transfers are. The *dma-get* command brings data from the main memory, (potentially through a shared L3 cache if exists), but if the copy of the same data is present in any of the L2 caches in the system, then the DMA engine can get and invalidate data from the appropriate cache. Invalidation in caches also happens when DMA write data to the main memory. So, coherency is maintained on the DMA transfer level which is much less frequent than on the memory access level. This relaxation in the memory coherency model is one of the reasons why multi-cores with local memories scale better than multi-cores with caches.

Additionally, for a fine grain maintaining of the memory consistency in the multi-core systems, DMA engines support atomic DMA transfers that work in a read-modify-write manner. This kind of transfers are inefficient in terms of performance since the process of reading the data, modifying it, and writing back to the main memory must be repeated until no failure in the atomicity is produced. Programmers usually try to avoid this kind of transfers, but as the architectural support it is a necessity in the multi-core systems, since exclusive access to a portion of the main memory must be guaranteed somehow, even in the architectures with on-chip local memories.

### 1.3.2 Programmability

All mentioned advantages of local memories are very desirable in a many-core CMPs. The problem is that local memories impose hard programmability difficulties - programming on a memory transfer level by hand coding every data transfer. Local memories impose a particular execution model that consists of the following four phases:

1. **Bringing the data to the local memory**. Essentially, programmer has to allocate buffer in the local memory and to program DMA transfer to bring data in. In order to do so, programmer has to assign the buffer in the local memory to the data in the main memory and to provide source address, destination addresses, and a tag for the transfer to the DMA engine. But besides programming DMA transfers, programmer has to take care about possible memory aliasing when assigning buffers to memory references (sometimes buffers are shared by few references in the code).

2. **Synchronization with the DMA transfers**. In this phase, programmer has to synchronize with the DMA transfers that bring data to be used in the computation. It must be ensured that the data transfers are finished prior to using the data in the local memory. Synchronization is done according to set of tags that programmer has to provide. So, programmer has to maintain time and place of the synchronization.

3. **Actual computation**. In this phase, data from the buffers is fetched in the functional units and after the appropriate computation is performed, output data is written back to the output buffers in the local memory. If the functional units are Single Instruction Multiple Data (SIMD) units, which are the most common multimedia extensions today. then additional programming challenges appear since SIMD units impose data alignment constraints and, moreover, use data arranged by a programmer in the local memory. So, in contrast to cache based architectures, here programmer can always try to do better when arranging the data in the local memory in order to achieve better application performance and improve SIMD coding by satisfying SIMD alignment constrains as much as possible. Dealing with alignment constraints is an important issue.

4. **Flushing the modified data from the local memory**. In this phase, all output buffers must be flushed back to the main memory. When flushing modified data in the multi-core environment, it is important to maintain memory consistency in order to ensure that the main memory contains the correct data when all cores finish flushing the data. Again, this is something that programmer has to do.

If the whole workload cannot fit in the local memory then the workload data is divided in smaller chunks that can fit in the local memory and the mentioned four steps repeat until the all chunks are processed. At the end, for the sake of efficiency, it is important to implement every step optimally (involing as litle of the additional code as possible) and to schedule DMA transfers efficiently in order to achieve good overlap of the communication and the computation (usualy done by using double buffering techniques). In Figure 1.6, we outline the main programming issues mentioned in this section that are encountered when

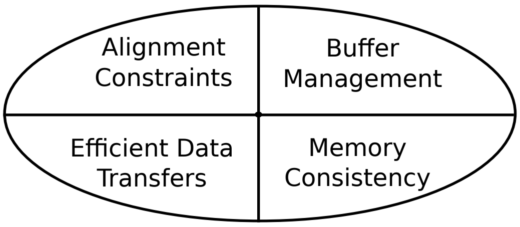


Figure 1.6: Programmability issues.

programming applications for on-chip local memories. We summarize all programmability issues into four categories:

- **Buffer management**. This is mostly related to the first phase of the execution model and the actions taken there: buffer allocation, buffer mapping, assignment of references, handling of aliasing, and DMA programming.
- **Memory consistency**. This issue occurs in the forth phase of the execution model when modified data has to be written-back to the main memory.
- **Efficient data transfers**. This issue targets overlapping communication with computation and mostly targets to minimize synchronization overhead present in the second phase of the execution model.
- **Alignment constraints**. Architectures with on-chip local memories and DMA engines have heavy alignment constraints imposed in the DMA engines. We observe that inability of doing unaligned DMA transfers can prevent programmers of freely dealing with buffer management for on-chip local memories which can additionally interfere with some alignment constraints imposed in the functional units, such as alignment constraints in SIMD units.

## 1.4 Motivation

Nowadays, it is desirable to write a code in a high level programming models, such as OpenMP, and to have a compiler and runtime system that can solve all mentioned programmability issues and port successfully the high level code to the underlying architecture, in our case to the architecture with on-chip local memory. This kind of compiler-based solutions are often difficult to deploy due to the lack of sufficient information at compile time to generate correct and efficient code. One of the main aims of this thesis is to develop such a runtime system that can help solving programmability issues (outlined in Figure 1.6) for the on-chip local memories.

On-chip local memories have been present in the computer architecture for a while, especially in the embedded systems domain. So, there are many techniques about handling programmability issues for on-chip local memories. Here, we select one particular work which is the most representative one in terms of the state-of-the-art techniques for handling programmability issues in local memories. That work is about IBM's compiler for Cell architecture [37]. Our interest is focused on two key techniques that IBM's compiler uses to handle programmability of on-chip local memories. Those two techniques are:

- **Tiling with static data buffering**. This techniques completely relies on compiler. It is aimed for references exposing stride access pattern (*regular* references). All buffer management in this approach is done at compile time. Since here memory aliasing must be handled at compile time then sometimes complex compiler analyses is required by this technique and this is the main drawback of tiling with static data buffering. The main advantage of this approach is the low control code overhead and the ability to work with big chunks of data in the computation phase of the execution model.

- **Compiler-controlled software cache**. In contrast to the static data buffering, this techniques relies on runtime systems. It is aimed for references exposing irregular access pattern (e.g.: pointer accesses). All buffer management is done at runtime. Buffers are organized as cache lines in the local memory and explicit data transfers are hidden via emulation of a hardware cache in software. IBM's compiler implements a 4-way set-associative software cache with 128-byte cache lines using a relaxed consistency model maintained through a dirt bits structure. Size of the cache storage is a configurable to the next sizes: 8KB, 16KB, 32KB, 64KB, and 128KB. Emulation of the hardware caches in software enables memory aliasing to be solved dynamically at runtime by doing look-up in software cache. The main advantage of this technique is that it is easy in terms of compiler analysis since the only thing compiler has to do is to guard every memory reference by control code responsible for all the actions which are typical for a cache, namely look-up, placement/replacement, data transfers, synchronization, address translation, and consistency. The main drawback of IBM's software cache is that it results in high overheads due to the dedicated control code that surrounds every global memory reference. In the rest of the thesis we refer to the IBM's software cache as *traditional software cache (TSC)*.

To illustrate the performance gap between tiling with static data buffering and traditional software cache, we compile STREAM kernels [67] with IBM XLC v10.1 compiler for the Cell BE [37] and execute on the Cell processor utilizing 8 SPEs. On X-axis in Figure 1.7, we have four STREAM kernels (Copy, Scale, Add, and Triad) and on Y-axis we have the obtained bandwidths from the executed STREAM kernels (theoretical maximal bandwidth in Cell is 25.6 GB/s, for details we refer to Appendix B). Two configurations are presented per kernel. One configuration corresponds to kernels compiled to use only tiling with static data buffering. We refer to this configuration as TIILING. The other configuration corresponds to the compilation that instruments a code to use only traditional software cache. We obtain the best performance in the tested kernels when configuring size of the traditional software cache to 64KB. We refer to this configuration as TSC. In Figure 1.7, we can see a huge gap in performance which shows the overhead present in the traditional software cache approach. Obtained bandwidths in kernels working with traditional software cache are more than ten times lower than bandwidths obtained in the kernels working under tiling with static data buffering. STREAM kernels are the best examples where tiling with static data buffering can give the best performance. Overhead in the traditional software cache approach is for a level of magnitude higher than in the tiling approach.

Traditional software cache and tiling in IBM's compiler are used complementary. Traditional software cache is a reliable approach used by default and tiling with static data buffering is an optimization used to improve performance. Compiler always tries to maximize the usage of tiling with static data buffering in order to get the best performance from the applications. In order to see how efficient this is in the real applications, we conduct an experiment on a set of NAS parallel benchmarks [12]. We compare the execution times with and without tiling optimization in architecture with on-chip local memories (SPEs in Cell BE). Additionally we compare performance obtained in the architecture with on-chip local memories against a SMP architecture. For this purpose, we use PowerPC 970MP processor [1] as a powerful SMP used in large High Performance Computing (HPC) infrastructures, and built in the same technology (90 nm) as the Cell processor. Comparing Cell programmed as a SMP to PowerPC 970MP is an immediate comparison to a modern SMP of the time when the Cell processor was launched. By no means this comparison is intended to evaluate and compare both architectures and generate architectural conclusions on whether which architecture is outperforming the other and in what cases. The main purpose is to quantify the obtained performance in the architecture with on-chip local memories where memory management is done in software (tiling and traditional software cache) in contrast to a completely hardware-based SMP architecture. We use a comparison to SMP architecture (PowerPC 970MP) in this thesis as a merit for competitiveness in performance of techniques for dealing with programmability issues in architectures with on-chip local memories.

In Figure 1.8 we present obtained execution times for IS, CG, FT, and MG applications

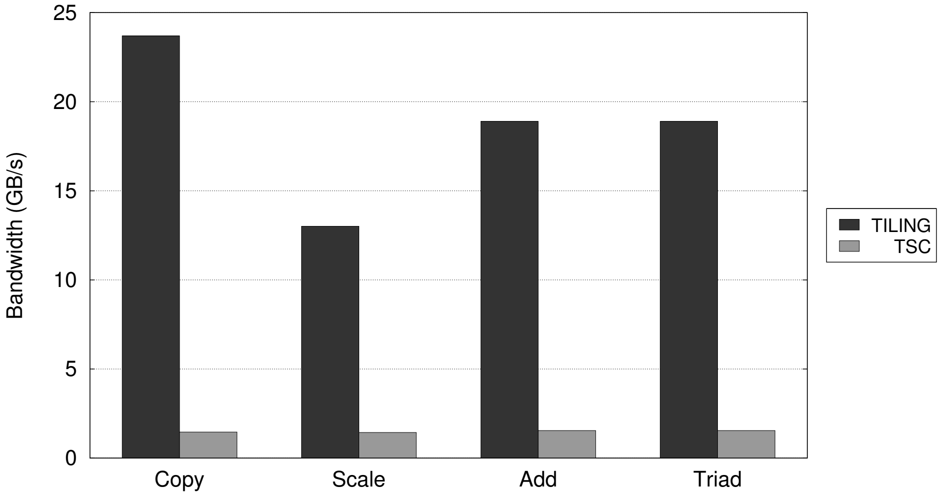


Figure 1.7: Overhead of the traditional software cache approach in respect to tiling with static data buffering approach in STREAM kernels.

executed under three different configurations. The first two configurations are TSC and TSC-TILING, both compiled by IBM XLC v10.1 compiler for Cell, and executed utilizing all 8 SPEs in the Cell processor. In the TSC configuration compiler is instrumented to use only traditional software cache. Size of the tradional software cache is configured to 64KB since we obtain the best performance for that size in the tested applications. In the rest of the thesis we always use this size for the traditional software cache. The TSC-TILING configuration corresponds to the default compilation where traditional software cache is used only at places where compiler fails to apply tiling optimization. The last configuration is PPC970MP, which is executed in PowerPC 970MP processor. This configuration is compiled by IBM XLC v10.1 compiler for SMP architectures.

We can see that tiling optimization improves performance in all tested applications. In some cases performance is improved more than 4 times (IS and FT). PowerPC 970MP performs better in CG, FT and MG while in IS we can see that execution time obtained in Cell processor is improved from being 2.5 times slower than PowerPC 970MP when traditional software cache is used, to being 1.7 times faster than PowerPC 970MP when tiling with static buffering is used. In the rest of the tested applications (CG, FT, and MG), PowerPC 970MP does better. The IS application is the simplest among the tested applications. It has four simple kernels where only one reference exposes irregular access pattern and requires usage of traditional software cache. Due to successful usage of tiling optimization in this application, good performance is obtained in the total execution time. We can see that it is even better than the execution obtained in PPC970MP configuration. In the rest of applications, picture is a bit different. Due to compiler restrictions, tiling is not applied at many places in CG, FT, and MG applications and then traditional software caches is used even for references which expose stride access pattern. For instance, in

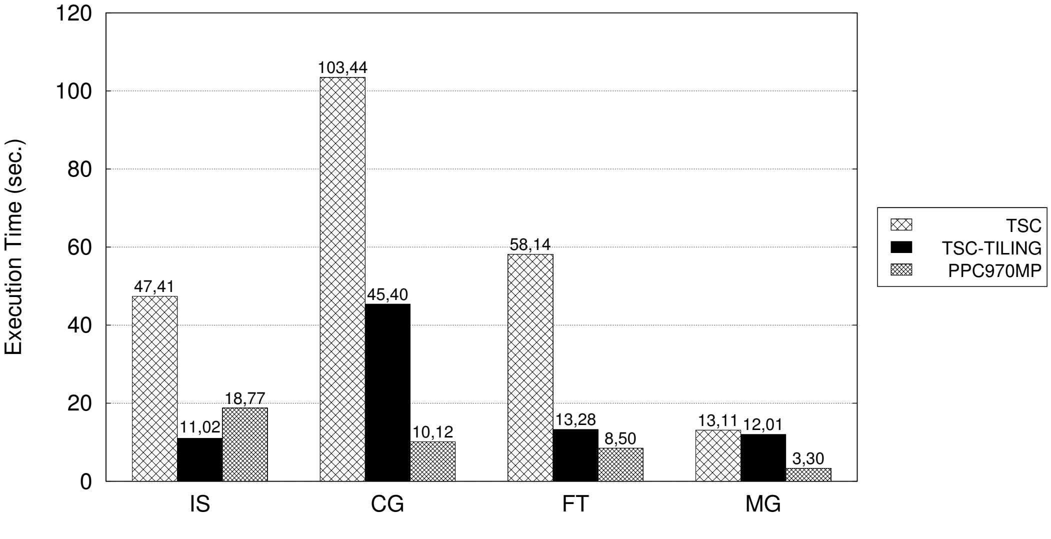


Figure 1.8: Comparison of IS, CG, FT, and MG execution times in the architecture with on-chip local memories (Cell BE) with execution times in PowerPC 970MP processor.

some cases possible memory aliasing appears between different memory references and the compiler is not able to determine a correct buffer allocation and tiling optimization fails. This is especially noticeable in MG application which is full of stride access references but with a plenty of references which are aliasing. Here tiling is applied in very few cases which results in poor improvements in the total execution time. We can see that TSC-TILING configuration in MG application is doing just 1.10 times better than TSC configuration. On the other side, FT application is also full of stride access references and we can see that TSC configuration is accelerated more than 4 times which makes TSC-TILING configuration to be competitive to PPC970MP configuration. The CG application suffers from irregular accesses which are treated by traditional software cache which is not competitive to the execution in SMP architecture.

In conclusion, the analyzed state-of-the-art technique for dealing with programambility issues in architectures with on-chip local memories could be optimized. A potential for performance improvements is in optimizing software caching and in relaxing compiler restrictions.

## 1.5 Contributions

The potential for the performance improvements of the anlyzed state-of-the-art technique for dealing with architectures with on-chip local memories, outlined in the previous section, is the main source of motivation for our first contribution which is the optimized runtime design and implementation of the *hybrid software cache architecture* that is in the middle of tiling with static data buffering and traditional software cache. Our aim is to optimize the way irregular memory references are treated, and for regular references to take good sides of tiling, such as low overheads and working with big buffers, but in general to unify the view of the software cache architecture in the applications in order to keep the simplicity of the required compiler analysis and bridge the compilers gap between tiling with static data buffering and software caching.

Our first contribution about hybrid software cache motivates our second contribution which is about addressing some of the programmability issues in Figure 1.6 through *hardware optimizations*. We propose a novel hardware for DMA engines that can better anticipate the needs of the execution model for on-chip local memories.

For each of two contributions we have few technical achievements. Achievements that are related to software caching are:

- The design and implementation of the novel hybrid access-specific software cache is the first and the main technical achievement of the first contribution of this thesis. Our approach classifies at compile time memory accesses in two classes, high-locality and irregular. Then it steers the memory references toward one of two specific cache structures optimized for their respective access pattern. The specific cache structures are optimized to enable high-level compiler optimizations to aggressively unroll loops, reorder cache references, and/or transform surrounding loops so as to practically eliminate the software cache overhead in the innermost loop. This achievement addresses mainly buffer management and efficient data transfers issues in Figure 1.6. As the result of the work done for this achievement, the next paper has been published:

  - Marc Gonzalez, Nikola Vujic, Xavier Martorell, Eduard Ayguade, Alexander Eichenberger, Kathryn O'Brien, Kevin O'brien, Chen Tong, Zehra Sura, Tao Zhang, *"Hybrid Access-Specific Software Cache Techniques for the Cell BE Architecture"*, In Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT 2008), Toronto, Canada, October, 2008.

- The second technical achievement is adaptive and speculative memory consistency support for multi-core architectures with on-chip local memories. This approach presents a set of alternatives to smooth the impact of considerable overheads related to software mechanisms to maintain the memory consistency in software cache implementations for multi-cores. In our approach, a specific write-back mechanism is introduced based on some degree of speculation regarding the number of threads actually modifying the same cache lines. This technical achievement addresses memory consistency issue in Figure 1.6. One paper has been published for this work:

  - Nikola Vujic, Lluc Alvarez, Marc Gonzalez, Xavier Martorell, Eduard Ayguade, *"Adaptive and Speculative Memory Consistency Support for Multi-core Architectures with On-chip Local Memories"*, In Proceedings of the 22nd Annual workshop on Languages and Compilers for Parallel Computing (LCPC 2009), Newark, Delaware, US, October, 2009.

- The third technical achievement addresses enabling prefetch techniques for our hybrid software cache. The new cache design enables automatic prefetch for high-locality memory accesses and modulo scheduling transformations for irregular memory accesses. Addressing memory wall by overlapping of the communication and computation in our hybrid software cache is the main purpose of this achievement. Efficient data transfers in Figure 1.6 are addressed specifically by this work. Two papers have been published:

  - Nikola Vujic, Marc Gonzalez, Xavier Martorell, Eduard Ayguade, *"Automatic Prefetch and Modulo Scheduling Transformations for the Cell BE Architecture"*, IEEE Transactions on Parallel and Distributed Systems (TPDS), vol. 21, no. 4, pp. 494–505, April 2010.
  - Nikola Vujic, Marc Gonzalez, Xavier Martorell, Eduard Ayguade, "Automatic Pre-Fetch and Modulo Scheduling Transformations for the Cell BE Architecture", In Proceedings of the 21st Annual workshop on Languages and Compilers for Parallel Computing (LCPC 2008), Edmonton, Canada, August, 2008.

In the software caching contribution, we start from the initial design of the novel hybrid software cache and we analyse where the bottlenecks are in order to propose new optimizations for the software cache - that is how optimized memory consistency and prefetching proposals come up. We do not stop with software caching, we cover some other aspects of the architectures with on-chip local memories, such as the quality of the generated code and its correspondence with the quality of the buffer management in local memories, in order to improve performance of these architectures. Therefore, we run our research till we reach the limit in software and start proposing optimizations on the hardware level in order to improve performance and programmability of the architectures with on-chip local memories. Thus, the second contribution of this thesis is about hardware optimizations. Here we have two main technical achievements:

- The first achievement addresses alignment constraints imposed in architectures with on-chip local memories. We propose a hardware realignment unit in the DMA engine that breaks alignment constraints imposed on the DMA transfer level in order to give more freedom to programmers and compilers when organizing data in the on-chip local memories. This proposal is motivated by the fact that the code generated by the compiler to exploit SIMD units suffers from significant memory alignment constraints which increases the code complexity as, in the general case, the compiler cannot be sure of the proper alignment of data. For that, the ISA provides either unaligned memory load and store instructions, or a special set of instructions to perform realignments in software. So, we propose a hardware realignment unit that takes advantage of the DMA transfers needed in architectures with on-chip local memories. While the data is being transferred, it is realigned on the fly by our realignment unit, and stored at the desired alignment in the local memory. This mechanism can help programmers to better organize data in the local memory so that SIMD units can possibly access the data with no special instructions. Finally, the data is realigned properly also when put back to main memory. Alignment constraints issue mentioned in Figure 1.6 is addressed by this work. We have published two papers related to this technical achievement:
  - Nikola Vujic, Felipe Cabarcas, Marc Gonzalez, Alex Ramirez, Xavier Martorell, Eduard Ayguade, *"DMA++: On the Fly Data Realignment for On-Chip Memories"*, IEEE Transactions on Computers, vol. 61, no. 2, pp. 237–250, February 2012.
  - Nikola Vujic, Marc Gonzalez, Felipe Cabarcas, Alex Ramirez, Xavier Martorell, Eduard Ayguade, *"DMA++: On the Fly Data Realignment for On-Chip Memories"*, In Proceedings of the 16th International Symposium on High-Performance Computer Architecture (HPCA-16), Bangalore, India, January 2010.
- The second technical achievement is a novel DMA engine that embeds the functionality of cache into the DMA engine and applies aggressive optimizations using novel hardware. We notice that even highly optimized hybrid software cache handlers sometimes result in high overheads (around 30% of the execution time) and then we propose to map some functionalities of our software cache into DMA engine in order to accelerate buffer management for on-chip local memories and test the limit of such approach. This proposal is an optimization addressing mainly buffer management issue in Figure 1.6. Related to this achievement, one paper has been published:

- Nikola Vujic, Lluc Alvarez, Marc Gonzalez, Xavier Martorell, Eduard Ayguade, *"DMA-circular: an Enhanced High Level Programmable DMA Controller for Optimized Management of On-chip Local Memories "*, In Proceedings of the 9th ACM International Conference on Computing Frontiers (CF'12), Cagliari, Italy, May 2012.

## 1.6 Structure

The structure of the thesis is as follows:

- Chapter 2 describes applications and methodology used to evaluate our contributions.
- Chapter 3 describes hybrid access-specific software cache, our first technical achievement of the first contribution of this thesis.
- Chapter 4 addresses the adaptive and speculative memory consistency support for our initial design of the hybrid software cache from Chapter 3.
- Chapter 5 is about efficient data transfers. Here, we describe prefetching and modulo scheduling transformations for overlapping communication and computation.
- Chapter 6 describes our first achievement of the second contribution of this thesis. Breaking alignment constraints on a DMA transfer level by doing realignments on the fly is described in this chapter.
- Chapter 7 is about our second hardware optimization where we explain the overheads of the software solution, and propose mapping of many software caching actions to the DMA engine.
- Chapter 8 concludes the thesis and presents some future work.

All technical chapters (Chapter 3, Chapter 4, Chapter 5, Chapter 6, and Chapter 7) are organized as follows. At the beginning of each chapter, we motivate the work and then we present its technical details. At the end, we present the evaluation of the proposal followed by the related work section and some conclusions.

# Chapter 2

# Methodology

This chapter describes the essential methodology and frameworks used for the evaluation of the proposals in this thesis. The organization of the chapter is as follows. Applications used for the evaluation are described in Section 2.1. Section 2.2 describes methodology for the real executions done in the thesis, while Section 2.3 introduces two frameworks used for the simulation of the hardware optimizations in this thesis.

## 2.1 Applications

For the evaluation purposes, we use the following applications in this thesis:

- **IS**, **CG**, **FT**, and **MG** - parallel applications from the NAS benchmark suite [12], which are parallelized using OpenMP directives. These four applications belong to the scientific computation domain: IS does an integer sort, CG is a conjugate gradient algorithm, FT computes a Fourier transformation, and MG realizes 3-dimensional multigrid relaxation with periodic boundary conditions. The most of the loops found in these applications operate on data structures organized as vectors, two or three dimensional matrices, accessed in most cases with stride accesses and in few cases under irregular access patterns. All of these applications can be used under few classes. In this thesis, we use class B of IS application. CG application is used under class B and shortened iteration space of 5 iterations, instead of 75 which are originally defined by class B of this application. We use shortened iteration space due to long execution time of CG application. Class A is used for FT and MG.

- **STREAM**: A simple synthetic benchmark that measures computation rate and sustainable memory bandwidth (in GB/s) for four simple vector kernels [70].

- **RandomAccess**: A simple synthetic benchmark that measures the rate of integer random updates or memory (GUPs) [67].

- **Matmul**: A blocked matrix multiplication, implemented with the kernel from Daniel Hackenberg [47].

- **SparseLU**: A blocked LU decomposition, that computes $L$ and $U$ and checks if $A = L \times U$ up to a certain accuracy [86].

- **Cholesky**: A blocked Cholesky factorization using the kernels by Alfredo Buttari [18].

- **Kmean**: The k-means clustering [58].

- **Knn**: The k-nearest neighbors algorithm [49].

In the evaluation section of each proposal, we list the applications which are used for the evaluation. In general, we always use IS, CG, FT, and MG applications, but for some proposals, we accompany these four applications with more applications in order to cover the wider range of applications when needed for better evaluation.

## 2.2 Real Executions

Real executions are used for the evaluation of the first contribution of this thesis. Two processors are used for the real executions:

- Cell, and

- PowerPC 970MP.

### 2.2.1 Executions on the Cell processor

All executions regarding the usage of on-chip local memories are performed on the Cell processor [44, 52, 56, 59]. This processor is the first-generation Cell Broadband Engine Architecture (CBEA) processor which is a hybrid multicore, built in 90nm technology, comprised of a PowerPC Processor Element (PPE), and eight Synergistic Processor Elements (SPEs). SPEs are accelerator cores that are equipped with on-chip local memories. Due to importance of the Cell processor for this thesis, we present more details about Cell processor in Appendix B.

All measurements are performed on a QS21 Cell BE blade [5] with two Cell processors running at 3.2 GHz with 1 GB of memory (512 MB per processor) under Linux Fedora release 7 (Kernel 2.6.22-5.fc7) and IBM Cell SDK 3.0. Only one Cell BE processor is used for the evaluation and all of eight SPEs are utilized. In all measurements, we exclude PPE from executing parallel regions in the tested applications. Unless explicitly mentioned, our results always use all 8 SPEs.

We use IBM XLC V10.1 compiler for the Cell BE [37] to compile programs for the executions. In general, we compile two different applications' code versions with this compiler in the thesis:

- One code version corresponds to TSC and TSC-TILING configurations (already introduced in Section 1.4), and this code version is compiled by single source compiler (cbexlc) which is capable of instrumenting SPE code to work with traditional software cache or, potentially, to use tiling with static data buffering optimization. The TIILING configuration, which is used only with STREAM application, is the same as TSC-TILING configuration. We refer to it as TIILING since we know that tiling with static data buffering optimization is completely applied in STREAM kernels and no traditional software caching is used.

- The other version of the applications' code corresponds to a code transformed to work with our hybrid software cache. Required compiler transformations are described in the appropriate sections (Sections 3.3, 4.3, and 5.3) of the chapters related to the hybrid software cache. We do not implement single source compiler for our code transformations. For the evaluation purposes, all code transformations are done manually in accordance to the described compiler requirements and transformations. Usually, transformed code consists of a PPE and SPE source code files which require separate compilation (ppuxlc, and spuxlc) and embeding of a SPE binary to a PPE binary (ppu-embedspu) in order to get the final CBEA executable.

Both versions of the code are compiled with the same level of optimization (-O5), without simdization (-qhot=nosimd) and including basic loop unrolling (-qunroll=auto). As we are conscious of the expensive execution of branch instructions on the Cell BE, we investigated loop unrolling of the innermost loops in the tested applications. We did not achieve any significant improvement (from 1% to 3%). Slight speedup was obtained by unrolling the loops up to 8 times, while for unrolling factor of 16 we obtained slower execution time than for unrolling factor 8. That is why in the used methodology not very aggressive loop unrolling performed by native XLC compiler (-qunroll=auto) is used. Regarding TSC-TILING configuration, two buffers are used for multibuffering, and the size of the static buffers for tiling optimization is adjusted to cause similar size of the DMA transfers as we have in the hybrid software cache.

### 2.2.2 Executions on the PowerPC 970MP processor

The PowerPC 970MP processor [1] is 64-bit Power Architecture dual-core processor built in 90nm technology with L2 cache of 1MB per core. It is designed to run on clock speeds between 1.2 and 2.5 GH with a maximum power usage of 75 W at 1.8 GHz and 100 W at 2.0 GHz.

All measurements are performed on a blade server type JS21, with two PowerPC 970MP processors running at 2.3 GHz with 8 GB of shared memory, under SUSE Linux Enterprise Server 10 (Kernel 2.6.16.60-perfctr-0.42.4-ppc64). Only one PowerPC was used for the evaluation. For the compilation, we use IBM XLC V10.1 compiler for SMP architectures under the highest level of optimization (-O5). In this thesis, measurements for PPC970MP configuration corresponds to the executions in this processor.

## 2.3 Simulations

Simulations are used for the evaluation of the second contribution of this thesis where hardware optimizations are proposed. Two different simulation infrastructures are used for the evaluation of the second contribution of this thesis. Those simulation infrastructures are:

- TaskSim, and
- PTLSim.

### 2.3.1 TaskSim

TaskSim [92] is a trace-driven simulator for multicore architectures with on-chip local memories, and cache-based multicore architectures. TaskSim provides cycle-accurate simulations of memory transfers: DMA transfers, or transfers initiated by a cache hierarchy. Thus, TaskSim models the memory system in a cycle-accurate mode: the DMA engine, the cache-hierarchy, the on-chip Interconnection Network, the Memory Interface Controller, the DRAM channels, and the DIMMs.

Three main simulation modes, supported in TaskSim, are:

- **Inout Mode**. This mode provides precise simulation of multicore architectures with on-chip local memories. It is based on the idea that, in the distributed memory architectures, the computation time on a given processor does not depend on what is happening in the rest of the system. Such is the case of architectures with on-chip local memories, where each processor's computation is based on data available in its private local memory. Nothing that happens on another processor can have an impact on the computation time. Execution time does depend on inter-thread synchronization, and the memory system: how long does it take to finalize a DMA transfer determines how long the processor will be waiting for data. Thus, TaskSim provides cycle-accurate simulations of DMA transfers in this mode, while it does not model the processors themselves. It relies on the computation time to be recorded in the trace in order to measure the delay between memory operations (starting / synchronizing with a DMA transfer) or inter-processor synchronizations (modeled as blocking semaphores). Also, events for starting / synchronizing with a DMA transfer are recorder in the trace.

- **Mem Mode**. This mode is for multicore cache-based architectures. Here, computation time is not recorded in the trace. Instead, instruction trace is provided to the simulator and execution time is completely based on executing only memory instructions.

- **Instr Mode**. In addition to the Mem Mode, this mode is enriched with processing all instructions in the instruction trace. Even executing the complete instruction trace, this mode does not model processor's pipeline in detail. Speed of executing the complete instruction trace is based on the instruction issue bandwidth limited by a dependency check for memory instructions. The core architecture model is intentionally simplistic, so that the modeling effort is devoted to the memory system.

In general, TaskSim is aimed for a cycle-accurate profiling on a memory transfer level, while cycle-accurate simulations of functional units and pipelines is out of the context of this simulator. In this thesis, we use TaskSim in Inout Mode for accurate simulations of DMA transfers. In this thesis, all traces for TaskSim are collected by executing applications on the Cell processor in the CELL BE blade described in Section 2.2.1. Also, the applications used for the trace gathering are hand-coded and compiled by XLC V10.1 compiler for the Cell BE under the highest optimization level (-O5).

For the details about simulator and for the accuracy and how the simulator is calibrated we refer to the TaskSim technical report [92], and some recently published materials regarding the usage of TaskSim [90, 91].

### 2.3.2 PTLsim

PTLsim [121] is a cycle accurate full system x86-64 microprocessor simulator. PTLsim models a modern superscalar out of order x86-64 compatible processor core. It is able to model processor core at a high level of detail and configurability. It provides highly detailed RTL-level model of all pipeline structures (e.g., issue scheme, branch predictor, functional units, register file). In addition, all microcode, the complete cache hierarchy, and I/O devices are modeled with true cycle accuracy. PTLsim runs unmodified 32-bit x86 and 64-bit x86-64 applications without special compilers.

PTLsim used in this thesis includes Wattch [17] library for gathering power and energy results from the simulations. Wattch is an architectural simulator that estimates CPU power consumption and the power consumption of its components. Power estimate used by Wattch in this thesis is based on a per-cycle resource usage counts generated through cycle-accurate simulations of the PTLsim.

In contrast to TaskSim, PTLsim is much slower, especially for simulations of the multicore architectures, due to a high level of simulation details. In this thesis, we use PTLsim for simulations where cycle-accurate profiling of the instruction pipeline is needed. All the applications, that are used by PTLsim in this thesis, are compiled by GCC compiler version 4.4.3 under the highest optimization level (-O3).

For more details about PTLsim, we suggest reading of the introductory paper about PTLsim [121], and the PTLsim technical report [120].

# Chapter 3

# Software Caching

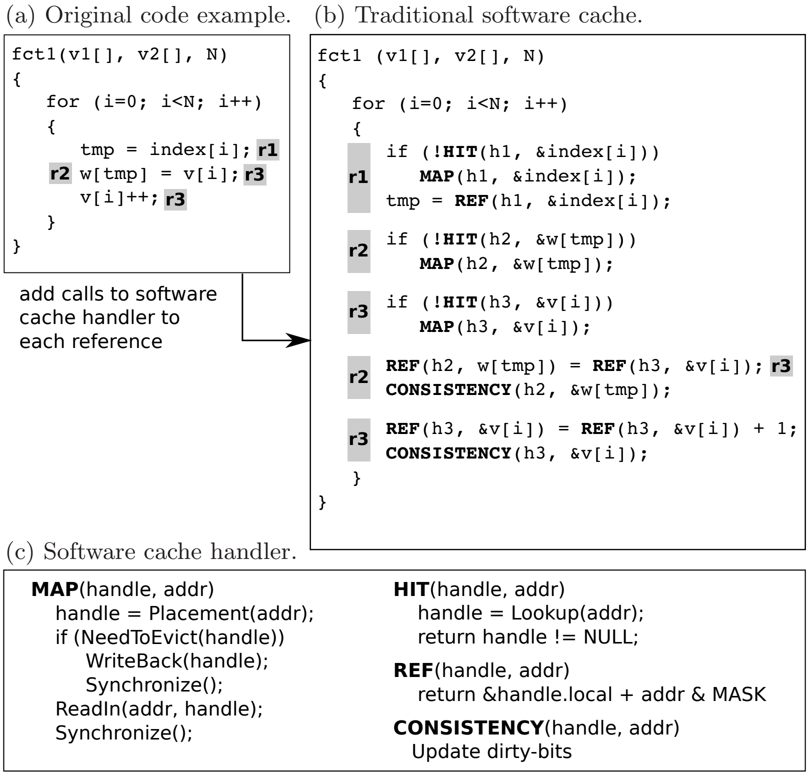
This chapter is about hybrid access-specific software cache. We analyze traditional software cache architectures and motivate our proposal in Section 3.1. Then, we present detailed design of our hybrid software cache in Section 3.2. Along with the novel software cache, new code transformation is needed and it is described in Section 3.3. Section 3.4 evaluates our approach using NAS benchmarks. We present related work in Section 3.5 and conclude the chapter in Section 3.6.

## 3.1 Motivation

As mentioned in the previous chapter, traditional software caching imposes high overheads in contrast to tiling with static data buffering. In this section we want to get more insights into those overheads and into a potential space for optimizations. In that manner, we analyse here a code transformation used by traditional software caching.

Consider the code example in Figure 3.1. The original code in Figure 3.1(a) shows three references, *index[i]*, *v[i]*, and *w[tmp]*. Assuming the arrays in global memory, Figure 3.1(b) depicts the same code with all of the required calls to a traditional software cache handler. Before each reference *r1*, *r2*, and *r3*, we need to check if the data is resident in the software cache (using the HIT macro). When not present, we call the miss handler (using the MAP miss handler). As shown in Figure 3.1(c), the miss handler MAP first locates a suitable cache line to evict, possibly writing the evicted line back to global memory, and then loads the requested line. Once the data has arrived (i.e. after a synchronization or blocking DMA), the data can be accessed, using the REF macro, in read or write mode. Following the *r2* and *r3* references, a consistency operation is performed to maintain relaxed consistency across all processors in the system. Consistency operations typically consist of updating dirty bits on write-back caches, or data communication on write-through caches. In our example, updating of memory consistency structures is associated to the CONSISTENCY handler.

All the software handlers surrounding references in Figure 3.1c are the reason for high overheads imposed by traditional software cache. Clearly, the transformed code in Figure 3.1(b) is far from optimal, especially for a high spatial-locality reference such as *index[i]*



(a) Original code example.

```
fct1(v1[], v2[], N)
{
   for (i=0; i<N; i++)
   {
      tmp = index[i]; r1
   r2 w[tmp] = v[i]; r3
      v[i]++; r3
   }
}
```

(b) Traditional software cache.

```
fct1 (v1[], v2[], N)
{
   for (i=0; i<N; i++)
   {
      if (!HIT(h1, &index[i]))
 r1      MAP(h1, &index[i]);
      tmp = REF(h1, &index[i]);

 r2   if (!HIT(h2, &w[tmp]))
         MAP(h2, &w[tmp]);

 r3   if (!HIT(h3, &v[i]))
         MAP(h3, &v[i]);

 r2   REF(h2, w[tmp]) = REF(h3, &v[i]); r3
      CONSISTENCY(h2, &w[tmp]);

 r3   REF(h3, &v[i]) = REF(h3, &v[i]) + 1;
      CONSISTENCY(h3, &v[i]);
   }
}
```

(c) Software cache handler.

```
MAP(handle, addr)
   handle = Placement(addr);
   if (NeedToEvict(handle))
      WriteBack(handle);
      Synchronize();
   ReadIn(addr, handle);
   Synchronize();

HIT(handle, addr)
   handle = Lookup(addr);
   return handle != NULL;

REF(handle, addr)
   return &handle.local + addr & MASK

CONSISTENCY(handle, addr)
   Update dirty-bits
```

Figure 3.1: Code operating with software cache approache.

and *v[i]* with $i = 0 \ldots N$. In tiling with static data buffering, *index[i]* and *v[i]* references are brought in big chunks in the local memory data is freely and efficiently accessed without any control code handlers surrounding memory accesses. In contrast to that, traditional software cache imposes overheads by executing HIT, MAP, and potentially CONSISTENCY handlers per each memory access. The scheme of emulating hardware caches in software and the execution of the trasformed code in Figure 3.1(b) provides too few guarantees to enable compiler optimizations that could significanntly lover software cache overheads. However, some good sides of tiling could be applied in software caching since for *high-locality* references, it is trivial to compute the number of useful data present in the current cache line. In other words, we can easily compute the number of loop iterations for which the current cache line can provide data for such references. Given such a number of iterations without a miss, we can iterate over these computations without any further software cache overhead. Also, it would be desirable to have a big cache line size in order to maximize the number of iterations that could be executed with no need of any cache intervention. However, needed code transformation is not possible with a traditional software cache interface. There are two main reasons why it is not possible to apply mentioned optimization using the traditional software cache interface:

- First, we must be able to pin a cache line in the software-cache storage, releasing it only when all high-locality references are done with it.

- Second, the cache must have at least one cache line per distinct high-locality reference in the loop, if we want to remove all checking code from the innermost loop.

Due to the listed requirements, software cache design has to be changed in order to enable some additional control over the geometry of the cache.

The code in Figure 3.1(b) is also suboptimal with respect to the second reference, *w[tmp]*, where *tmp* is equal to the original value of *index[i]*. This access pattern corresponds to an indirect access, which typically exhibit very poor data locality. Because of this *irregular* access pattern, the data is unlikely to be found in the cache. Performance can only be achieved by exercising as many irregular accesses as possible in parallel (i.e. without synchronization or blocking DMA) for maximum communication overlap. Again, traditional caches interfaces are not suitable, as they typically provide only a small set-associativity, which directly limits the number of concurrent irregular accesses. Also, since no spatial reuse is expected, typical (long) cache lines are likely to waste bandwidth and thus slow down the execution.

Our proposal is to design a hierarchical, hybrid software-cache architecture that is designed from the ground up to enable compiler optimizations that reduce software-cache overheads. We identify two main data access patterns, one for high-locality and one for irregular accesses. Because the compiler optimizations targeting these two patterns have different objectives and requirements, we have designed two distinct cache structures that best respond to these distinct access patterns and optimization requirements. In particular, our design includes:

- A high-locality cache with a variable configuration, fully associative scheme, lines that can be pinned, and a sophisticated eager write-back mechanism

- A transactional cache with fast, fully associative lookup, short lines, and an efficient write-through policy.

Performance evaluation indicates that improvements due to the optimized software-cache structures combined with the proposed code transformations translate into 3.5 to 8.4 speedup factors, compared to a TSC approach (Section 1.4) for a set of parallel NAS applications. In some applications our hybrid approach performs as good as TSC-TILING approach (Section 1.4) or even better achieving competitive performance in respect to SMP architectures such as PowerPC 970MP.

## 3.2 Software Cache Design

We describe in this section the design of our hierarchical, hybrid software-cache. Figure 3.2 shows the high level architecture of our software cache. Memory references exposing a high degree of locality are mapped by the compiler to the High Locality Cache, and the others, irregular accesses are mapped into the Transactional Cache. The Memory Consistency Block implements the necessary data structures to maintain a relaxed consistency model according to the OpenMP memory model [51].

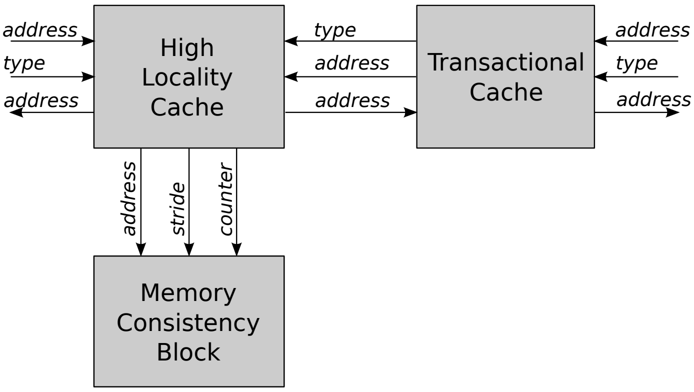


Figure 3.2: Block diagram of our software cache.

The cache is accessed through one block only, either the High Locality Cache or the Transactional Cache. Both caches are consistent with each other. The hybrid approach is hierarchical in that the *Transactional Cache* is forced to check for the data in the High Locality Cache storage during the lookup process.

### 3.2.1 The High Locality Cache

The High Locality Cache enables compiler optimizations for memory references that expose a high degree of spatial locality. The main characteristics are:

- The ability to pin cache lines using explicit reference counters
- Deliver good hit ratios
- Full associativity
- Maximize the overlap between computation and communication.

#### 3.2.1.1 High Locality Cache Structures

The High Locality Cache is composed of the following six data structures, depicted in Figure 3.3:

- The Cache Storage to store application data.
- The Cache Line Descriptors to describe each line in the cache
- The Cache Directory to retrieve the lines
- The Cache Unused List to indicate the lines that may be reused
- The Cache Translation Record to preserve for each reference the address resolved by the cache lookup

- The Cache Parameters to record global configuration parameters.

The Cache Storage is a block of data storage organized as $N$ cache lines, where $N$ is total cache storage divided by the line size. The line size is described by the Cache Line Size parameter, and must be a power of 2. Note that the storage overhead for control structures (metadata such as the Cache Line Descriptors, the Cache Directory, and the Cache Unused List) is directly proportional to the number of cache lines. For an architecture specific implementation purposes, it is important to estimate the space that can be used for the metadata and according to that to determine the minimum line size to be supported (the minimum line size determines the maximum number of cache lines to be supported for a specific cache storage size).

Each cache line is associated with a unique Cache Line Descriptor that describes all there is to know about that line. The *Global Base Address* is a global memory address that corresponds to the base address associated with this line in global memory. Its *Local Base Address* corresponds to the base address of the cache line in the local-memory cache storage. Its *Cache Line State* records state such as whether the line holds modified data or not. Its *Reference Counter* keeps track of the number of memory references that are currently referencing this cache line. Its *Directory Links* is a pair of pointers used by the cache directory to list all of the line descriptors that map to the same cache directory entry. Its *Free Links* is a pair of pointers used to list all the lines that are currently unused (i.e. with reference counter of zero). Its *Communication Tags* are a pair of integer values used to synchronize data transfers to/from the software cache. For synchronization we use DMA tags. Number of distinct DMA tags to be used for synchronization depends on the number of DMA tags supported in the architecture itself (the specific numbers related to our implementation for the Cell BE architecture are present in Section 3.4.1).

The Cache Translation Record preserves information generated by the lookup process and to be later used when data is accessed by the actual reference. It contains 3 elements; the global base address of the original reference, the local base address in the cache storage, and a pointer to the cache line descriptor.

We implement an efficient, fully associative cache structure using the Cache Directory structure. It contains a sufficiently large number of double-linked lists (we denote to it as $M$ double-linked lists), where each list can contain an arbitrary number of cache line descriptors. A hash function is applied to the global base address to locate its corresponding list, which is then traversed to find a possible match. The use of a hash function enables us to efficiently implement cache configurations with up to $M$-way fully associative caches. Basically, we have a fully associative cache with a lookup speed of $M$-way set associative caches. In practice, this number $M$ is limited by the allowed metadata size, since more double-linked lists there are, more storage overhead has to be introduced for maintaining the heads of $M$ double-linked lists.

The Cache Unused List is a double-linked list which contains all the cache line descriptors no longer in use. Other cache parameters include parameters such as the *Cache Directory Hash Mask*, a mask used by the cache directory to associate a global base address with its specific linked list.

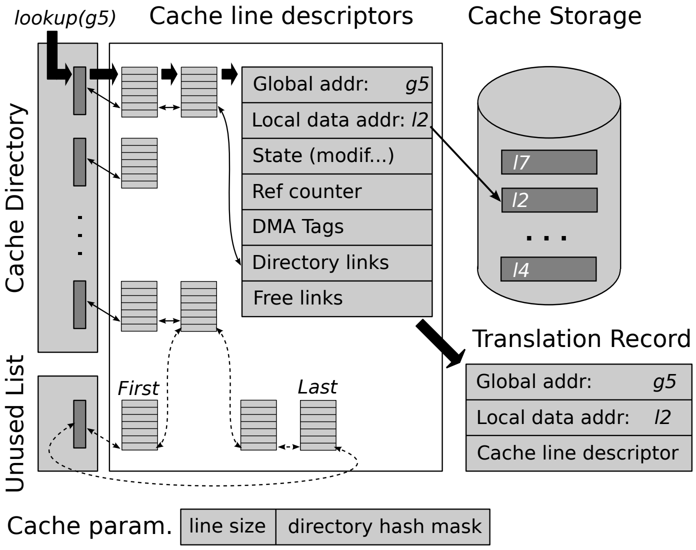


Figure 3.3: Structures of the High Locality Cache.

#### 3.2.1.2 High Locality Cache Operational Model

The operational model for the High Locality Cache is composed of all the operations that execute upon the cache structures and implement the primitive operations shown in Figure 3.1(c), namely *lookup, placement, communication, synchronization and consistency mechanisms*. The following paragraphs describe each type of operation.

The *lookup* operation for a given reference $r$, translation record $h$, and global address $g$ is divided in two different phases. The first phase checks if the global address $g$ is found in the cache line currently pointed to by the translation record $h$. When this is the case, we have a hit and we are done. Otherwise, we have a situation where the translation record will need to point to a new cache line in the local storage. The lookup process then enters its second phase. The second phase accesses the cache directory to determine if the referenced cache line is already resident in the cache storage. When we have a hit, we update the translation record $h$ and we are done. Otherwise, a miss occurred and we continue with placement and communication operations.

The reference counter is often updated during the lookup process. Whenever a translation record stops pointing to a specific cache line descriptor, the reference counter of this descriptor is decremented by one. Similarly, whenever a translation record starts pointing to a new cache line descriptor, the reference counter of this new descriptor is incremented by one.

The placement code is invoked when a new line is required. Free lines are discovered when their descriptors reference counter reaches zero. Free lines are immediately inserted at the end of the unused cache line list. Modified lines are then eagerly written back to global memory. When a new line is required, we grab the line at the head of the unused cache line list after ensuring that the communication performing the write-back is completed, if the line was modified.

We support a relaxed consistency model. While it is the Memory Consistency Block responsibility to maintain consistency, the High Locality Cache is responsible for informing the consistency block of which subsets of any given cache line have been modified and how to trigger the write-back mechanism. Every time a cache line miss occurs, cache thus informs the Memory Consistency Block about which elements in the cache line are going to be modified.

The communication code performs all data transfer operations asynchronously. For a system such as the Cell BE processor with a full-featured DMA engine, we reserve a half of the full available range of DMA tags for data transfers from main memory to the local memory, and the other half for data transfers in the reverse direction. In both cases, tags are assigned in a circular manner. Tags used in the communication operations are recorded in the *Communication Tags* field of the Cache Line Descriptor. All data transfers tagged with the same DMA tag are forced by the DMA hardware (using DMA fences) to strictly perform in the order they were programmed.

The synchronization operation is supported by the data in the Cache Line Descriptor, in the *Communication Tags* field. The DMA tags stored in this field are used to check if any pending data transfer is completed. The *Communication Tags* record every tag that invokes the corresponding cache line.

When accessing data, the global to local address translation is supported through the translation record. The translation operation is composed of several arithmetic computations required to compute the reference's offset in the cache line and to add the offset to the local base address.

### 3.2.2 The Transactional Cache

The Transactional Cache is aimed at memory references that do not expose any spatial locality. Due to expected high miss ratios, this cache is designed to deliver very low hit and miss overhead while increasing concurrency and offering high throughput that can enable a good overlap of computation and communication. It supports a relaxed consistency model using a write-through policy. The design introduces very simple structures that allow supporting for *lookup*, *placement*, *communication*, *consistency*, *synchronization*, and *translation* mechanisms.

#### 3.2.2.1 Transactional Cache Structures

The Transactional Cache is composed of the following four data structures, shown in Figure 3.4:

- The Cache Directory to retrieve the lines.
- The Cache Storage to hold the application data.
- The Translation Record to preserve the outcome of a cache lookup for each reference.
- Some additional cache state.

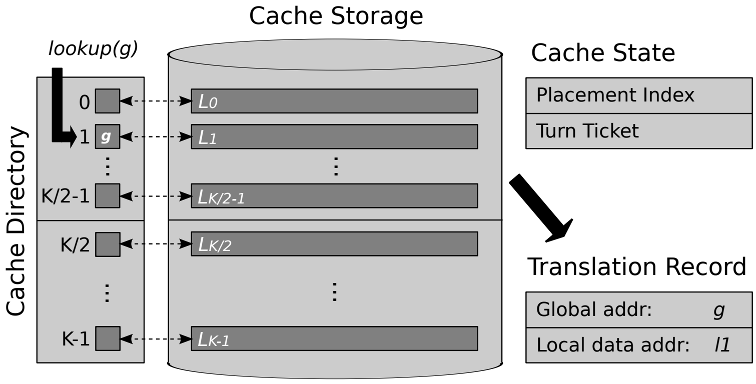


Figure 3.4: Structures of the Transactional Cache.

The Cache Directory is organized as a vector of $K$ 4-byte entries. Each entry holds the global base address associated with this entry's cache line. The index of the entry in the directory structure is also used as index into the Cache Storage to find the data associated with that entry. The directory entries are packed in memory and aligned at the boundary that enables the use of fast SIMD compares to more quickly locate entries. As we will see in the evaluation section (Section 3.4), the performance overhead of the Transactional Cache is very important since its handlers are invoked per each irregular memory access. Thus, designing directory on a way to enable fast SIMD compares serves the purpose of making fully associative lookup in the transactional cache as fast as possible. For instance, on platforms where 4 entries fit into one SIMD register, such as the SPEs, we perform a $K$-way address match using $K/4$ compare SIMD instructions. Obviously, $K$ should not be too big number in order to have reasonably quick lookup time. As explained in Section 3.2.2.2, we bound the number of cache lines by the number of DMA tags due to consistency issues.

To increase concurrency, the Cache Directory and storage structures are logically divided in two equal-size partitions; the *Cache Turn Ticket* indicates which partition is actively used. Within the active partition, the *Cache Placement Index* points to the cache line that will be used to service the next miss.

At a high level, the active partition is used to bring in the cache lines required by the current transaction, while the other partition is used to buffer the cache lines of the prior transaction while their modified data is being written back to global memory.

In order to achieve high throughput, we use small lines for the Transactional Cache in order to minimize communication time, rather than trying to capture already poor locality of the irregular access patterns by bigger cache lines. Thus, we set the cache line size for the Transactional Cache to be the smallest transfer size that can efficiently use the available bus bandwidth.

#### 3.2.2.2 Transactional Cache Operational Model

In this thesis, a transaction is a set of computation and related communication that will happen as a unit (but never rollback). Operations in a transaction happen in four consecutive steps:

1. Initialization.
2. Communication into local memory.
3. Computation associated with the transaction.
4. Propagation of any modified state back to global memory.

During initialization, in Step 1, the *Cache Turn Ticket* is flipped to point to the other partition. The *Cache Placement Index* is set to the first cache line of the new active partition. In our configuration, its value is either 0 or $K/2$ when the ticket is, respectively, pointing to partition 0 or 1. In addition, all the cache directory entries in the new active partition are erased.

In Step 2, the data corresponding to each global-memory reference is brought into the local memory, using sequences of lookup and possibly calls to the misshandler. The lookup process for a given reference $r$, translation record $h$, and global address $g$ first proceeds with a standard High Locality Cache lookup, since we do not want to replicate data in both cache structures, and if miss occurs then it proceeds with lookup in the Transactional Cache. This first lookup can be avoided if address $g$ can be guaranteed not to be found in the High Locality Cache. When a hit occurs, the *Local Base Address* field in translation record $h$ is simply set to point to the appropriate sub-section of the line in the High Locality Cache storage. When a miss occurs, however, we proceed by checking the address $g$ against the entries in transactional cache directory. As mentioned in Section 3.2.2.1, this lookup is fast on architectures with SIMD units, such as in the Cell BE SPEs [45]. When a miss occurs, a placement operation is executed. When a hit occurs, the lookup can operate in one of two ways. If the hit occurred within the active partition, we simply update the transaction record $h$. If, however, the hit occurred within the other partition, we need (for simplicity) to migrate the line to the active partition; a placement operation is used for this operation as well.

The placement code simply installs a new directory entry and associated cache line data at the line pointed by the *Cache Placement Index*. The placement index is then increased by one (modulo $K$). Communications generated by the miss in Step 2 results into an asynchronous line size transfer into local memory.

Step 3 proceeds with the computation, using the same translation record as seen in Section 3.2.1.

In Step 4, every modified storage location that was modified by a store in Step 3 is directly propagated (write through) into global memory. This approach to relaxed consistency eliminates the need for any extra data structures (such as dirty bits) and do not introduce any transfer atomicity issue. These asynchronous communications occur regardless of whether a hit or miss occurred in Step 2. Moreover, only the modified bytes of data (not the entire line, unless the entire line was modified) are transferred into global memory during Step 4.

In order to ensure consistency within and across transactions, every data transfer is tagged with the index of the cache line being used, and a fence is placed right after the data transfer operation. Due to the fence, all data transfers tagged with the same tag are forced by the hardware to perform strictly in the order under which they were programmed. This bounds the maximum number of cache lines to the number of available tags, but it makes the communication tags management trivial with very low overhead. The synchronization code occurs in precisely two places. The first synchronization is placed between Steps 2 & 3, to ensure that the data arrive before being used. When Partition 0 is active, we wait for data transfer operations with tags $[0 \ldots \frac{K}{2} - 1]$, and wait for tags $[\frac{K}{2} \ldots K - 1]$ otherwise. For the data transfer initiated in Step 4, the synchronization code is placed at the beginning of the next transaction with the same value for the *Cache Turn Ticket*, synchronizing with the data transfer operations tagged with numbers $[0 \ldots \frac{K}{2} - 1]$, or $[\frac{K}{2} \ldots K - 1]$.

### 3.2.3 Memory Consistency Block

The Memory Consistency Block (MCB) is designed to maintain a relaxed consistency model, for memory references mapped into High Locality Cache, in accordance to the OpenMP memory model [51]. The block includes the necessary data structures to support the write-back operation, that is, the transfer of a modified cache line from the local memory to main memory. It corresponds to a traditional implementation based on the use of dirty-bits and atomicity. More detailed work on optimizing MCB is presented in Chapter 4.

#### 3.2.3.1 Standard Write-Back

The eviction process is executed at the hardware cache line level, as atomicity is supported in the DMA engine at this data granularity. The hardware cache line size is usually 128 bytes, therefore, software cache lines are evicted in chunks of that size: every 128 bytes within a cache line are read, then modified data is merged according to the dirty-bits information and finally the 128-bytes are sent back to main memory atomically. If the atomic data transfers fails, the whole process is repeated until no failure is produced. The data structure required to support this mechanism is the Local Dirty Bits structure that keeps track of modified parts of the resident cache lines. This metadata structure is allocated in the local memory of every core and it is an important storage overhead.

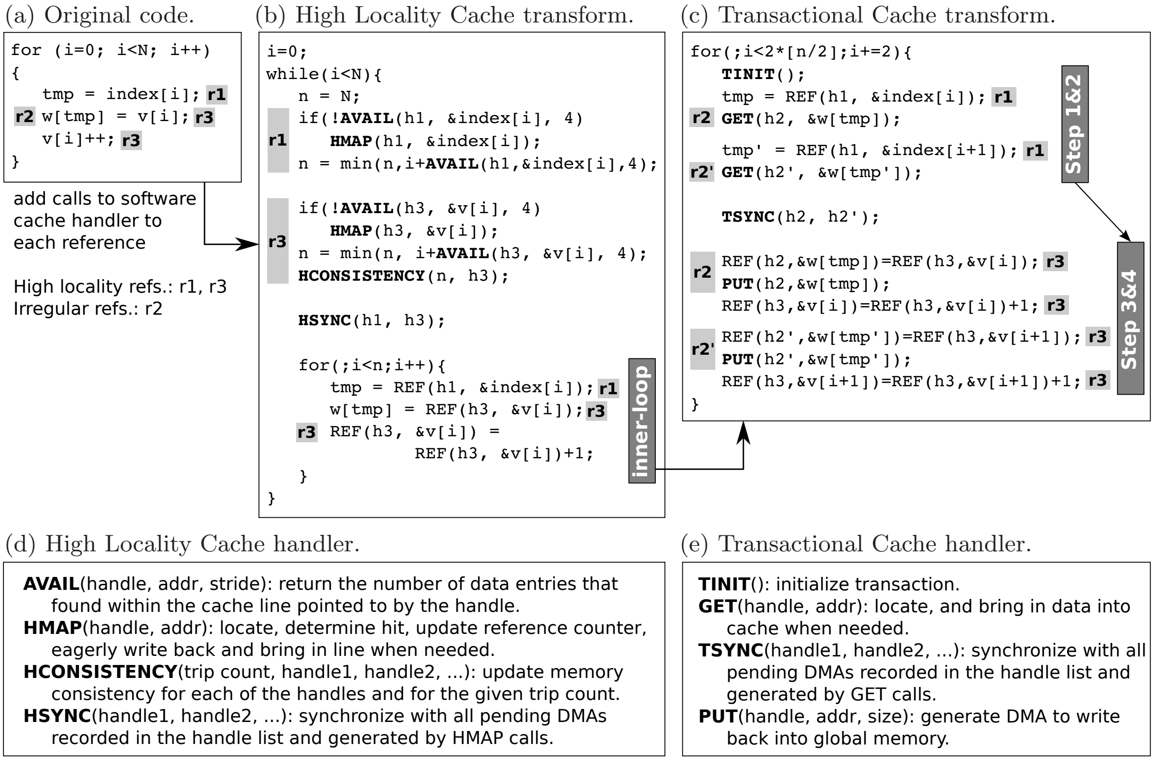
## 3.3 Initial Code Transformations

We describe in this section the type of code transformation techniques that are now enabled using our hierarchical, hybrid software cache. It uses here the distinction between memory references with high-locality and irregular access pattern. Accesses with high-locality patterns are mapped to the High Locality Cache and all irregular accesses are mapped to the *Transactional Cache*. With no loss of generality, the code transformation targets the execution of loops.

The code transformations are performed in three ordered phases:

1. Classifying memory references into high-locality (regular) and irregular accesses

2. Code transformation to optimize high-locality memory references



(a) Original code.

```
for (i=0; i<N; i++)
{
   tmp = index[i]; r1
r2 w[tmp] = v[i]; r3
   v[i]++; r3
}
```

add calls to software cache handler to each reference

High locality refs.: r1, r3
Irregular refs.: r2

(b) High Locality Cache transform.

```
i=0;
while(i<N){
   n = N;
r1 if(!AVAIL(h1, &index[i], 4)
      HMAP(h1, &index[i]);
   n = min(n,i+AVAIL(h1,&index[i],4);

r3 if(!AVAIL(h3, &v[i], 4)
      HMAP(h3, &v[i]);
   n = min(n, i+AVAIL(h3, &v[i], 4);
   HCONSISTENCY(n, h3);

   HSYNC(h1, h3);

   for(;i<n;i++){
      tmp = REF(h1, &index[i]); r1
      w[tmp] = REF(h3, &v[i]); r3
   r3 REF(h3, &v[i]) =
                REF(h3, &v[i])+1;
   }
}
```

(c) Transactional Cache transform.

```
for(;i<2*[n/2];i+=2){
   TINIT();
   tmp = REF(h1, &index[i]); r1
r2 GET(h2, &w[tmp]);

   tmp' = REF(h1, &index[i+1]); r1
r2' GET(h2', &w[tmp']);

   TSYNC(h2, h2');

r2 REF(h2,&w[tmp])=REF(h3,&v[i]); r3
   PUT(h2,&w[tmp]);
   REF(h3,&v[i])=REF(h3,&v[i])+1; r3

r2' REF(h2',&w[tmp'])=REF(h3,&v[i+1]); r3
   PUT(h2',&w[tmp']);
   REF(h3,&v[i+1])=REF(h3,&v[i+1])+1; r3
}
```

(d) High Locality Cache handler.

**AVAIL**(handle, addr, stride): return the number of data entries that found within the cache line pointed to by the handle.
**HMAP**(handle, addr): locate, determine hit, update reference counter, eagerly write back and bring in line when needed.
**HCONSISTENCY**(trip count, handle1, handle2, ...): update memory consistency for each of the handles and for the given trip count.
**HSYNC**(handle1, handle2, ...): synchronize with all pending DMAs recorded in the handle list and generated by HMAP calls.

(e) Transactional Cache handler.

**TINIT**(): initialize transaction.
**GET**(handle, addr): locate, and bring in data into cache when needed.
**TSYNC**(handle1, handle2, ...): synchronize with all pending DMAs recorded in the handle list and generated by GET calls.
**PUT**(handle, addr, size): generate DMA to write back into global memory.

Figure 3.5: Example of C code and its code transformation. Without losing generality, we assume data size of 4 bytes in the used arrays.

3. Code transformation to optimize irregular memory references.

We illustrate this process in Figure 3.5.

### 3.3.1 Classification of Memory Accesses

Reference analysis used in this phase is based on the algorithms presented in [81]. In this phase, memory accesses are classified as high-locality or irregular accesses. A strided access with constant stride less than the size of a cache line in the High Locality Cache is classified as a high-locality (regular) access. All the other accesses are classified as irregular accesses. For high-locality accesses, it is important that the stride between them is constant. The size of a stride is important in determining how many accesses can be done per cache line. If the size of a stride is bigger than the size of a cache line then it does not make sense to use High Locality Cache. If it is not possible to determine the size of a stride then we classify the related accesses as irregular. In terms of correctness, the code will not be inconsistent if we classify them as high-locality accesses, but in terms of performance and wasting of the cache storage it is better to classify them as irregular accesses.

The size of a cache line is a configurable parameter in our software cache and depends on the number of regular accesses determined during the classification of memory accesses. It is desirable to have cache lines as big as possible and in that sense we use a greedy strategy in order to determine the optimal size of a cache line. In the first attempt we set a size of a cache line to be the biggest value (as described in Section 3.4.1, in our configuration for Cell it is 4K). Then we classify memory accesses. Since we need as many cache lines as there are high-locality memory references in a code section, after classification we have to check if this is satisfied. If this condition is not satisfied then we downgrade the size of a cache line and repeat classification again. We repeat those steps until we find a correct match, which is possible as long as the number of high-locality memory references in a loop is less than the maximum number of the supported cache lines in the High Locality Cache. Otherwise, extra memory references are classified as irregular and mapped into Transactional Cache.

Figure 3.5(a) shows the classification of the references for our exemplary code, where memory accesses *index[i]*, and *v[i]* with $i = 0 \ldots N$ are labeled as high-locality or regular while memory access *w[tmp]* (with *tmp=index[i]*) is labeled as irregular memory accesss.

### 3.3.2 High-Locality Access Transformations

In Phase 2, we transform the original *for*-loop into two nested loops that basically perform a dynamic subchunking of the iteration space. As shown in Figure 3.5(b), the outer *while*-loop iterates as long as we have not visited all of the original $N$ iteration points. The inner *for*-loop iterates over a dynamic subset of iterations, $n$, where $n$ is computed as the largest number of iterations for which none of the high-locality references will experience a miss. As shown in the innermost loop in Figure 3.5(b), there is no cache overhead (REF overheads detailed in Figure 3.1(c) are essentially free).

We detail now the work introduced by each high-locality reference in the outer *while*-loop body. Consider in Figure 3.5(b) the work associated with reference *r1* with translation record *h1* and global memory address *g1=&index[i]*. First, we compute with the AVAIL macro the number of iterations for which address *g1* will be present in the cache line currently pointed to by its translation record *h1*. If this number is zero, we have a miss, and we invoke the HMAP miss handler. This miss handler performs as indicated in Section 3.2.1.2. Note that a miss does not imply communication, as the cache line may already be present in the cache due to other references. Once the new line is installed in the translation record *h1*, we recompute AVAIL and update the current dynamic sub-chunking factor $n$ so as to take into account the number of iterations for which reference *r1* will not experience a miss. Next high-locality reference (*v[i]*) is processed on the same way. Hence it is write access reference, it is additionally instrumented with the call to a HCONSISTENCY handler.

After processing all high-locality references, the dynamic sub-chunking factor $n$ is definitive, and can be used to inform the memory consistency block of all of the memory locations that will be touched by high-locality references in the inner *for*-loop. Note that all this work is performed in parallel with the asynchronous DMA requests possibly initiated by the HMAP miss handler.

The last operation is to perform synchronization on all pending DMAs, using the communication tags found in the cache line descriptors associated with each of the high-locality references.

An additional task is to compute the optimal High Locality Cache configuration. At the very least, we need one cache line per distinct high-locality memory reference, but a few more enables better latency hiding for the eager writeback process. Note, however, that we may always downgrade one or more high-locality memory accesses to be treated as irregular references. In practice, we select the largest line size that satisfies each of the high-locality memory references present in the loop.

### 3.3.3 Irregular Accesses Transformations

In Phase 3, we transform the inner *for*-loop to optimize cache overhead for irregular memory accesses. The first task is to determine the transactions. In our work, the scope of a transaction is a basic block, or a subset of. Large transactions are beneficial as they potentially increase the number concurrent misses, thus increasing communication overlap. In general, a transaction can contain as many distinct irregular accesses as there are entries in a single partition of the transactional cache (for instance, it is 16 entries in our configuration for Cell, as described in Section 3.4.1). Because of our focus on loops, larger transactions are mainly achieved through loop unrolling. In our example, we unrolled the inner *for*-loop by a factor of 2 (for conciseness) so as to include two *w[tmp]* and *w[tmp']* references within a single transaction.

The code generated for a transaction closely follows the four step process outlined in Section 3.2.2.2. As shown in Figure 3.5(c), we first initialize the transaction (Step 1) and then proceed in asynchronously acquiring the data of each irregular reference $r2$ and $r2$ using the GET macro (Step 2). Once all irregular references have been processed, we issue a TSYNC operation to synchronize on all pending DMAs issued by the GET operations. We then access the data using the REF macro (Step 3) and write back the modified data using the PUT macro (Step 4).

The code generated for a transaction closely follows the four step process outlined in Section 3.2.2.2. As shown in Figure 3.5(c), we first initialize the transaction (Step 1) and then proceed in asynchronously acquiring the data of each irregular reference *r2* and *r2'* using the GET macro (Step 2). Once all irregular references have been processed, we issue a TSYNC operation to synchronize on all pending DMAs issued by the GET operations. We then access the data using the REF macro (Step 3) and write back the modified data using the PUT macro (Step 4).

### 3.3.4 Summary of Code Transformations

To summarize the achievement of the code, we observe that there is no software cache overhead associated with high-locality memory accesses in the innermost loop (*for*-loop in Figure 3.5(c)). Second, all communications generated by high-locality memory access misses occur concurrently. Third, all communications generated by irregular memory access misses occur concurrently.

## 3.4 Evaluation

We present in this section an evaluation of the initial design of our hybrid software-cache approach. We first present implementation details of our hybrid software cache for Cell. Then, we investigate what are the main components that contribute to the significant speedup in the overall performance achieved by our cache design compared to the TSC approach and we study the overall performance and compare with TSC-TILING and PPC970MP configurations. Then, we analyze cache overhead distribution looking for potential places to be addressed by further optimizations. The CG, IS, FT, and MG applications are used for the evaluation under methodology described in Section 2.2.

### 3.4.1 Implementation parameters for Cell

Our implementation of the hybrid software cache is for the Cell BE architecture (See Appendix B for the details about the architecture). A small local memory of 256KB in SPEs is shared for code and data. In order to leave enough space for the code, we select the following parameters for the implementation of our hybrid software cache. The High Locality Cache storage is 64KB big, and it can store between 16 to 128 cache lines of sizes from 512 to 4K bytes. The *Cache Directory Hash Mask* is set to support 128 double-linked lists. Due to 32 distinct DMA tags in Cell BE, our communication tags thus range from *0* to *31*. In the High Locality Cache tags *0* to *15* are used for data transfers from main memory to the local memory, and tags *16* to *31* for data transfers in the reverse direction. The Transactional Cache storage is organized as a small 4KB capacity cache, with 32 128-bytes cache lines where each line is aligned at a 128-byte boundary to enable fast lookup with SIMD compares. The storage overhead for the metadata for both cache structures is around 45KB in total. Approximately, complete footprint of our implementation is 113KB (64KB + 4KB + 45KB) used for the software cache storage and control structures. The rest of the SPEs local memory (143 KB) is for code and stack data only.

### 3.4.2 Overall Performance

In this section, we evaluate the performance impact of the two major features that significantly contribute to higher performance. The first one is the ability of reducing the amount and complexity of the control code surrounding the global memory references; and the second one is the ability of overlapping communication and control code execution.

We study here two configurations of our hybrid approach. The first configuration is our cache prototype as described in Section 3.2, using all of the code transformations proposed in Section 3.3. We refer to this configuration as HYBRID thereafter. The second configuration is a modified version of HYBRID, where each data transfer is performed synchronously (blocking DMA) to prevent any overlap between computation and communication. We refer to this configuration as HYBRID-synch thereafter. Both configurations are compared against TSC, TSC-TILING and PPC970MP configurations introduced in Section 1.4.

Figure 3.6 shows the execution time in seconds for the IS, CG, FT and MG applications. Let us first compare the HYBRID-synch and TSC designs. This comparison gives us an

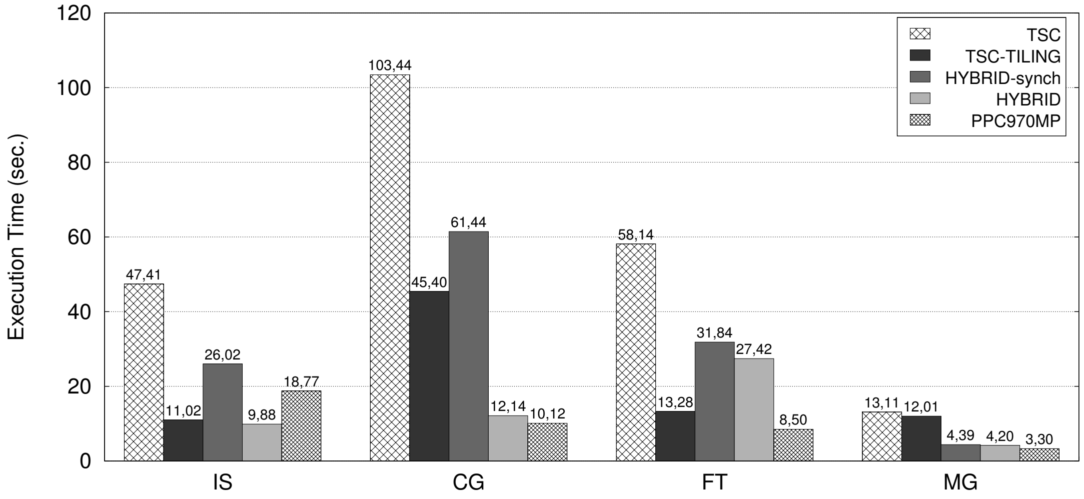


Figure 3.6: Performance of the hybrid software cache approach.

indication about which configuration has higher software-cache overhead as both configurations uses blocking (synchronous) DMA requests. Among the four applications, FT and MG, are highly dominated by regular memory accesses. For these two, large lines (4KB and 2KB) found in the High Locality Cache of the hybrid scheme significantly reduce the number of communications, compared to the 128B lines of the traditional software cache. Also, the hybrid approaches get rid of all lookup code within the innermost loop, whose trip count is roughly proportional to the size of 4KB or 2KB cache line; whereas the traditional approach has one lookup per global memory reference. This results in speedup factors of 1.83 for FT and 2.98 for MG over the entire applications.

The two other applications, CG and IS, are highly dominated by irregular memory accesses. We have measured that 90% of the execution time in CG and 70% of the execution time in IS is devoted to the execution around references steered to the Transactional Cache in the HYBRID-synch approach. So, these two applications are dominated by the activity in the Transactional Cache. When comparing the execution time of HYBRID-synch versus TSC achieved speedup factors are 1.82 for IS and 1.68 for CG. Interestingly, this speedup factors do not come from reduced miss ratios, as shown in Table 3.1. Indeed, since both configurations use blocking DMA, the reduced execution time can only be explained by faster software cache primitives found in the hybrid configurations. Specifically, the write-through mechanism used by the Transactional Cache is far more efficient than a mechanism

Table 3.1: MISS ratios for the IS and CG applications

| Application | HYBRID | TSC |
|---|---|---|
| IS | 33.39% | 35.46% |
| CG | 17.29% | 17.16% |

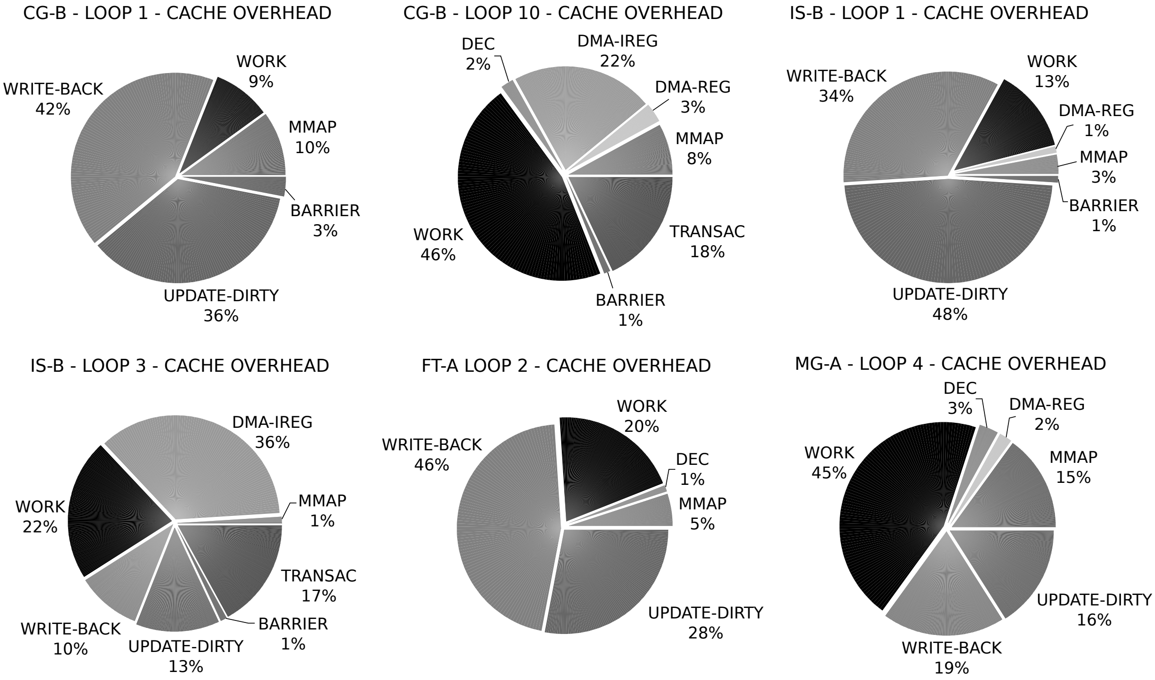
based on dirty bits and atomicity. Indeed, the dirty-bit mechanisms (such as those found in traditional scheme and in the High Locality Cache of the hybrid approach) require an atomic operation involving one DMA to read the line, a merge of the modified data, followed by a DMA to write back the modified line. In addition to these two DMA operations, the execution of the control code for the Dirty Bits structure is typically a highly branching code. None of these overheads are found in the Transactional Cache in the hybrid design.

Let us now compare, in Figure 3.6, the TSC and TSC-TILING configurations to the HYBRID configuration, which uses asynchronous DMA commands to enable overlapping communication and the control code execution. We now achieve speedup factors of 4.79 for IS, 8.52 for CG, 2.12 for FT and 3.12 for MG in respect to TSC configuration. Not surprisingly, the applications dominated by irregular memory references benefit much more from the asynchronous communications, as the transactional cache and associated code transformations attempt to maintain up to 32 DMA misses in flight. In respect to TSC-TILING, we obtain speedups of 1.11 for IS, 3.73 for CG, 2.85 for MG, while for FT, HYBRID configuration is two times slower than TSC-TILING. The HYBRID configuration does better when the memory references in the loop computations cannot be safely treated by the tiling transformation because of compiler restrictions. CG and IS applications are dominated by irregular memory accesses that are treated more efficiently by Transactional Cache than traditional software cache. In MG application, memory aliasing appears between different memory references and the compiler is not able to determine a correct buffer allocation, as mentioned in Section 1.4. In those situations the compiler generates inefficient code that uses traditional software cache which is not as efficient as HYBRID configuration using High Locality Cache in regular applications such as MG application. On the other side, TSC-TILING successfully applies tiling with static data buffering in FT application, which makes HYBRID not to be competitive to TSC-TILING in this applications due to low overheads of the tiling with static data buffering approach. The HYBRID configuration uses long cache lines of the High Locality Cache in FT application, but some overheads still remain. For instance, memory consistency mechanism in the High Locality Cache maintains dirty bits information and uses it in a time of write-back, which tiling with static data buffering does not suffer from. In order to determine what should be further optimized in our hybrid software cache, the detailed evaluation of the overheads of the HYBRID configuration is presented in the next section.

In respect to PPC970MP configuration, HYBRID performs better only in IS application, being 1.90 times faster. In CG and MG, HYBRID is competetive to the PPC970MP being no more than 1.30 times slower. For FT, HYBRID is far behind PPC970MP and should be improved by factor of 3 in order to be competetive with PPC970MP.

### 3.4.3 Cache Overhead Distribution

Six loops are described in Figure 3.7 in terms of the cache overhead distribution. Loops are selected as the representative cases of what is generally observed in the tested applications. For applications dominated by irregular accesses (CG and IS) two loops are selected, one is the most time consuming loop from the application (loop 10 in CG, and loop 3 in IS), and the other is a loop dominated by regular accesses. For applications dominated by regular



**WORK**: time spent in actual computation.
**UPDATE-DIRTY**: time spent updating the dirty-bits info.
**DMA-REG**: time spent synchronizing the DMA data transfers in the high-locality cache.
**TRANSAC.**: time spent executing control code of the transactional cache.
**MMAP**: time spent in executing lookup, placement actions and DMA programming.
**WRITE-BACK**: time spent in the write back process.
**DMA-IREG**: time spent synchronizing the DMA transfers in the transactional cache.
**DEC.**: time spent in the pinning mechanism for the cache lines.
**BARRIER**: time spent in the synchronization barrier at the end of the parallel computation.

Figure 3.7: Cache overhead distribution for CG (loops 1 and 10), IS (loops 1 and 3), FT (loop 2), and MG (loop 4). Total execution time is broken down and percentages are displayed. Components of the execution time which correspond to less than 1% of the execution time are omitted from the charts.

accesses (FT and MG), the most time consuming loops are selected (loop 2 in FT, and loop 4 in MG).

Loop 1 in CG is an example of a loop doing initialization on plenty of references. This loop is dominated by the activity in the High Locality Cache. The actual loop computation in this loop takes only 9% of the total loop execution. Notice how the memory consistency support (write-back and update of dirty bits) corresponds to a considerable amount of overhead, around 78%, divided in to a 36% for updating the dirty bits information and 42% devoted to the write-back operation. This is due to high degree of write-access references in this loop, five out of six regular references require maintaining of the memory consistency. Notice that mapping a cache line to the High Locality Cache corresponds to 10% of the execution time which is more than the time spent in the computation phase. Synchronization on the barrier in this loop takes 3% of the execution. For the second case of CG (loop 10), the distribution is very different. This loop is dominated by the activity in the Transactional Cache and it is the most time consuming loop of CG. Here, the actual computation corresponds to a 46%, while 18% is devoted to control code on the Transactional Cache, plus a 22% of time devoted to synchronize with the DMA engine for transactional data transfers. The case of loop 10 in CG represents a case which does not suffer from memory consistency issues since in this loop there is only one regular write-access reference which is not as frequently accessed as the rest of references in this loop. The case of CG loop 10 is composed of two nested loops where mentioned regular write-access reference is preset at the outer level while the inner most loop contains reduction operation which modifies one scalar value. That's why write-back and update of dirty bits are omitted from the chart for this loop since both stand for close to 0% of the execution time in this particular loop.

On the other side, the case of loop 3 in IS is dominated by the activity in the Transactional Cache but contains some regular write-access references which are accessed as often as the irregular references in this loop (accesses at the same loop-nest level). We can see that write-back corresponds to 10% of the execution time while the update of dirty bits is around 13% of the execution time. Due to noticeable memory consistency components, actual computation corresponds to much less percentage than computation in loop 10 in CG. Here it is 22% in contrast to 46% observed in loop 10 in CG. Notice that only 1% is devoted to control code for High Locality Cache, while 17% is devoted to the Transactional control code in this loop with the significant time, 36% of loop execution, spent in synchronizing with transactional DMA transfers. The other case of IS is loop 1 which has no irregular memory accesses. In this looop situation is similar to the case of CG loop 1. Actual computation corresponds to only 13% of the total execution, while memory consistency takes 82% (48% for update of dirty bits, and 34% for write-back mechanism). In this loop, synchronization with DMA transfers takes only 1%.

The case of FT loop 2 consists of more complex kernels with plenty of references to be handled by the High Locality Cache. In this loop overhead of the memory consistency is as high as in simple kernels, such as CG loop 1 and IS loop 1. Notice, update of dirty bits and write-back mechanism take 28% and 46% of the loop execution respectively, while control code for High Locality Cache takes only 5% of loop execution. Work component corresponds to 20% of the execution.

Loop 4 from MG is the most time consuming loop in MG application. It is more complex kernel than CG and IS loops. Similarly to FT loop 2, MG loop 4 contains only high locality references but with a different degree of write-access references. In FT loop 2, 50% of all references treated through the High Locality Cache require maintaining of memory consistency. In a case of MG loop 4, that percentage is lower and equal to 17 (three out of 18 references are write-access references). Due to smaller degree of write-access references, memory consistency in this loop takes 35% of total execution time (16% for update of dirty bits, and 19% for write-back mechanism), in contrast to 74% in FT loop 2. Control code for the High Locality Cache takes 15% of the loop execution, while only 2% are devoted to synchronizing with DMA transfers.

In general, actual computation component is saturated by the cache overhead. Computation can be as little as 10% of the execution time in some loops. No matter the type of loop, whenever regular write-access references exist, memory consistency component (update of dirty bits and write-back) significantly contribute to low percentage of time devoted to the computation phase. This is drastic in a case of loops with no activity in the Transactional Cache since there memory consistency overhead can go above 80% of loop execution time. In loops dominated by the activity in the Transactional Cache, noticeable overheads are the synchronization with transactional DMA transfers and the control code for operating with Transactional Cache. So, even we achieved good overlap of communication with control code execution, some communication is still not overlapped enough imposing high synchronization overheads in the Transactional Cache. Synchronization with DMA transfers in High Locality Cache does not seem to be a problem by now.

## 3.5 Related Work

As analyzed in the evaulation, tiling transformations and static buffers may be used to reach the same level of code optimization [37]. In general, when the access patterns in the computation can be easily predicted, static buffers can be introduced by the compiler, usually involving loop tiling techniques. This approach, however, requires precise information about memory aliasing at compile time. When not available at compile time, one could still generate several code versions depending on runtime alignment conditions, at a cost in code size. Or, we can follow the approach in this chapter, where we postpone the association between static buffers and memory references until run time. In doing so, we solve all the difficulties related to memory aliasing since the High Locality Cache lines are treated as buffers that are dynamically allocated. Of course, if the performance of a software cache approach is to match that of static buffers, clearly, any efficient implementation should work with a cache line size similar to that of the static buffers (usually 1KB, 2KB, 4KB, depending on the number of memory references to treat) [21]. This is the case of the software cache design presented in this chapter.

Chen et al [22, 23, 64] have further developed an integrated static-buffer and software-cache approach so as to enable the parallelization of loops that include references directed to both structures. Their scheme specifically addresses software-cache references that may potentially use data currently present in static buffer, or vice versa. Their approach attempts to minimize coherency operations using a layered approach including compile time and runtime disambiguation.

S. Seo et al [98], propose a software cache architecture, with a fast and adaptive placement and replacement mechanism for accelerator-based architectures with local memories. In general, this solution defines simple mechanisms and compiler code transformations to smooth the overhead related to the local memory management. Proposed software cache architecture operates on the similar way as traditional software cache, using similar code transformation. This approach does not expose any aggressive optimizations such as those found in our proposal: hybrid design which highly optimizes accesses to high-locality references and maximizes overlapping of the communication with the control code execution.

Miller and Agarwal [71] proposed a software instruction cache. The main issue is how to treat the direct/indirect jumps. The proposed mechanisms, although falling into software caching techniques, are very different from the ones presented in this chapter. There is one particular aspect in common to our proposal. Their approach must be able to pin basic blocks in the instruction software cache while there might be jumps that point to them. Our solution is based on counters indicating how many memory references point to a cache line.

The work in HotPages/FlexCache [75, 76] is similar to our work but is not as powerful and simple as the work described in this thesis. It relies on mechanisms of considerable complexity and needs of a compiler analysis with some degree of accuracy (e.g: pointer analysis). In addition, it does not provide a solution where the compiler fully removes the checking code around memory references presumed to reference the hot pages.

Software caching techniques have been applied to reduce the amount of power consumption associated to cache management. These proposals face similar problems as the ones addressed in our work. For instance, Direct Addressed Caches [118] propose the elimination of the tag checks by making the hardware to remember the exact location of a cache line, so that hardware can access data directly. This proposal requires the definition of new registers in the architecture to relate load/store operation to specific cache lines, leaving to the compiler the decision of what memory references have to be associated to the additional registers. In addition, this proposal requires new load/store instructions to actually make use of the associated registers. The work in SoftCache [40] is based on binary rewriting, which requires a complete control-flow and data-flow analysis of binary images. Clearly this is not the case in the proposal of our work, which avoids such complexity.

In Udayakumaran et al. [110] data allocation in scratchpad memories is addressed. Although a different context, the problems solved are similar to the ones addressed in this chapter. The proposal is based on code region analysis through code profiling, plus frequency analysis of memory accesses. Derived from this information, a timestamp process is applied to basic blocks and specific points in the code are defined where control code is injected to move data between DRAM and SRAM. The solution is able to efficiently treat memory accesses that could be statically associated to a specific code point. Pointerbased accesses and aliasing are not totally solved, since the presented solution explicitly maintains a side structure (e.g: a height tree) for detecting if data is missing in the SRAM and ensure a correct address translation. The proposal in this chapter is generic enough to not rely on any profiling information for the compiler to statically allocate data. On the opposite, the proposal in this chapter delays at runtime the allocation and mapping decision, and succeeds in removing most if not all the control code surrounding the memory references.

## 3.6 Conclusions

This chapter presents a novel, hybrid access-specific software cache architecture that maps memory accesses according to the locality they expose. It presents a design of two distinct and custom cache structures that are tailored for two different kind of references: high-spatial locality and irregular references. Performance evaluation indicates that the hybrid design is more efficient than TSC-TILING approach in IS, CG, and MG applications achieving speedup factors in the 1.11 to 3.73 range. Achieved performance in these three benchmarks is also competetive to execution in the PowerPC 970MP processor. However, performance should be further improved since HYBRID is far enough from TSC-TILING and PPC970MP configurations in FT application.

Performance evaluation analyses overhead distribution for the hybrid software cache by giving insights on how execution time is distributed when hybrid approach is used. We outline two main overheads to be addressed. Evaluation shows that overhead related to the update of dirty bits and write-back for High Locality Cache can take up to 80% of the execution time. Memory consistency maintaining through dirty-bits is inherited from the traditional software cache without any optimizations in our initial design of the hybrid software cache. This overhead is a candidate to be addressed. Also, significant time spent in the synchronization with transactional DMA transfers denotes second important overhead to be addressed. In the next chapter we address memory consistency overhead and in Chapter 5 we address the synchronization overhead.

# Chapter 4

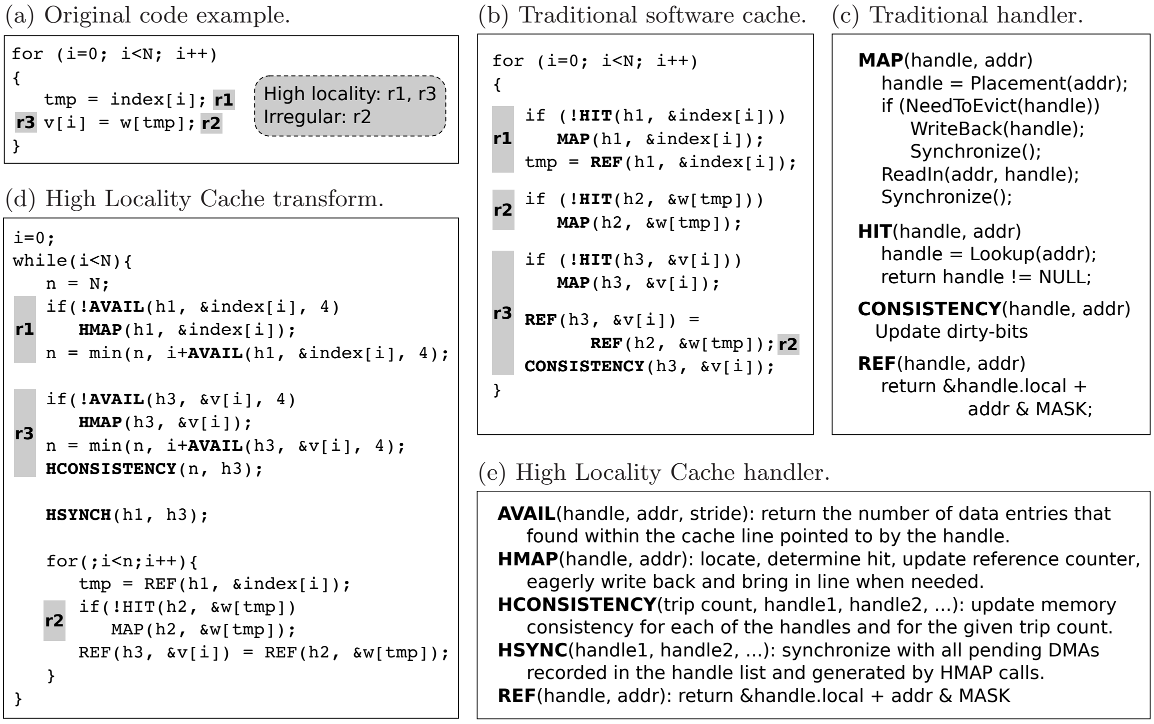
# Memory Consistency

This chapter is about adaptive and speculative memory consistency support for multi-core architectures with on-chip local memories. We analyze sources of memory consistency overheads in our hybrid-access specific software cache and motivate further optimization regarding memory consistency support in Section 4.1. Then, we present detailed design of the new memory consistency block in Section 4.2. Needed compiler support is explained as well, and Section 4.3 is devoted to it. Section 4.4 evaluates our proposal. Related work is presented in Section 4.5 and conclusions of the chapter are in Section 4.6.

## 4.1 Motivation

As it is shown in Chapter 3, hybrid access-specific approach can substantially accelerated code by first determining whether a reference exhibit a high-locality or an irregular access pattern, and then directing each reference to one of two custom cache structures that are tailored for their respective access pattern. Figure 4.1(d) shows the code transformation following this hybrid approach. Comparing the codes in Figure 4.1(b) and Figure 4.1(d), it is clear that the control code surrounding memory references *r1* and *r3* has been totally removed, and now this control code is executed at cache line level, not at memory reference level. This is not the case for the code responsible for memory consistency, as every store operation requires control to monitor the modified data. Thic control is done by CONSISTENCY macro in traditional software cache, and by HCONSISTENCY macro in our High Locality Cache. In this chapter, we address this source of overhead in the context of a cache design that enables the code transformation in Figure 4.1(d).

Software caching techniques rely on mechanisms that keep track of which elements in a cache line hold modified data. This is typically done using a dirty-bits data structure that is kept up-to-date by compiler introduced code. Every store operation is instrumented so as to record which bytes are being modified in the accessed cache line. The dirty-bits information is then used during the write-back process, so that only the modified parts of the cache line are transferred to main memory. Usually this implies reading the cache line, merging the modified data with the unmodified data, and then transferring the whole cache line back to main memory. The overall process suffers from two different sources of



(a) Original code example.

```
for (i=0; i<N; i++)
{
   tmp = index[i]; r1
r3 v[i] = w[tmp]; r2
}
```

High locality: r1, r3
Irregular: r2

(b) Traditional software cache.

```
for (i=0; i<N; i++)
{
   if (!HIT(h1, &index[i]))
r1    MAP(h1, &index[i]);
   tmp = REF(h1, &index[i]);

r2 if (!HIT(h2, &w[tmp]))
      MAP(h2, &w[tmp]);

   if (!HIT(h3, &v[i]))
      MAP(h3, &v[i]);
r3
   REF(h3, &v[i]) =
         REF(h2, &w[tmp]); r2
   CONSISTENCY(h3, &v[i]);
}
```

(c) Traditional handler.

```
MAP(handle, addr)
   handle = Placement(addr);
   if (NeedToEvict(handle))
       WriteBack(handle);
       Synchronize();
   ReadIn(addr, handle);
   Synchronize();
HIT(handle, addr)
   handle = Lookup(addr);
   return handle != NULL;
CONSISTENCY(handle, addr)
  Update dirty-bits
REF(handle, addr)
   return &handle.local +
              addr & MASK;
```

(d) High Locality Cache transform.

```
i=0;
while(i<N){
   n = N;
   if(!AVAIL(h1, &index[i], 4)
r1    HMAP(h1, &index[i]);
   n = min(n, i+AVAIL(h1, &index[i], 4);

   if(!AVAIL(h3, &v[i], 4)
      HMAP(h3, &v[i]);
r3 n = min(n, i+AVAIL(h3, &v[i], 4);
   HCONSISTENCY(n, h3);

   HSYNCH(h1, h3);

   for(;i<n;i++){
      tmp = REF(h1, &index[i]);
      if(!HIT(h2, &w[tmp])
   r2    MAP(h2, &w[tmp]);
      REF(h3, &v[i]) = REF(h2, &w[tmp]);
   }
}
```

(e) High Locality Cache handler.

**AVAIL**(handle, addr, stride): return the number of data entries that found within the cache line pointed to by the handle.
**HMAP**(handle, addr): locate, determine hit, update reference counter, eagerly write back and bring in line when needed.
**HCONSISTENCY**(trip count, handle1, handle2, ...): update memory consistency for each of the handles and for the given trip count.
**HSYNC**(handle1, handle2, ...): synchronize with all pending DMAs recorded in the handle list and generated by HMAP calls.
**REF**(handle, addr): return &handle.local + addr & MASK

Figure 4.1: Code transformation for traditional software cache design and High Locality Cache design. The assumed size of the data found in the used arrays is 4 bytes.

overhead:

- The number of executed instructions is increased in order to update the dirty-bits as well as to perform the merging operation.

- The transfer of the line from and to main memory result in additional communication for which we must further enforce atomicity to ensure correctness.

One interesting measurement is the number of cache lines that actually need the use of both dirty-bits and atomicity. Those cache lines are the ones that have been modified by more than one thread. In general, determining which cache lines are going to be modified by more than one thread is not a simple problem. This information is not available at compile time, and at run time, complex mechanisms have to be introduced to distribute the data describing which thread is modifying which cache line.

The simplest solution is to generally introduce atomicity and dirty-bits (this is used in the traditional software cache and our High Locality Cache inherited it from there), regardless of the fact that a cache line actually does not need both. This creates a significant loss of performance, most noticeable in the scientific domain where parallel loops are mostly composed of regular and predictable access patterns that translate into the fact that very few cache lines are modified by more than one thread. For instance, assume that the *i*-loop in Figure 4.1(a) is parallelized and the iteration space is cut in even chunks containing consecutive $i$-iterations. In such situation only the cache lines accessed within the borders of the assigned chunks of iterations are going to be modified by more than one thread. If the $i$-loop is executed by $N$ threads, only *N-1* cache lines might be concurrently modified, while all other cache lines are going to be modified by only one thread. This defines a significant disproportion that in the context of a fully software controlled environment, it is likely to incur in unacceptable overheads. It is necessary to provide the cache design with the ability to detect and adapt to particular access patterns that result in the situation previously described.

Our proposal is to design an adaptive and speculative memory consistency support in the context of a hybrid cache architecture. Adaptive and speculative memory consistency support is implemented in the Memory Consistency Block (presented in Chapter 3). The Memory Consistency Block is enhanced with new functionality and besides the standard write-back (Section 3.2.3.1) there are two new write-back mechanisms:

- Lock-Unlock Write-Back
- Speculative Write-Back.

Performance evaluation indicates that improvements due to the optimized memory consistency translate into speedup factors from 1.30 to 2.60, comparing hybrid software-cache with optimized memory consistency to the initial design of the hybrid software-cache with standard write-back.

## 4.2 Enhanced Memory Consistency Block

The main characteristic of the modified design is that of implementing a speculative write-back mechanism for specific cache lines. The enhanced Memory Consistency Block (MCB) defines one heuristic to classify cache lines and according to this classification some cache lines are decided to be treated speculatively, while others are not. The basis for this classification is the definition of *memory regions* and an associated *access pattern* among the executing threads.

### 4.2.1 Cache Line Classification

The adaptive and speculative behavior of the MCB is controlled by information collected at runtime. This information describes memory regions and how they are supposed to be accessed by the executing threads. Once this information is obtained (usually at the beginning of the execution), its main purpose is to guide MCB to behave in a particular way throughout the rest of the execution. It is important to bear in mind that memory regions are supposed to be useful only for High Locality Cache, which suffers from memory consistency overheads. Therefore, memory regions are specified only for a data that is exclusively modified by the High Locality Cache.

Memory regions are specified by three parameters. Two parameters are starting and ending addresses of the memory region while the third parameter is a description of the access pattern within the memory region. The access pattern within a memory region is described as a data distribution: the memory region is cut in even *chunks*, and chunks are assigned to cores according to some pre-determined distribution schemes. The supported schemes try to capture the most common access patterns defined by parallel loops that operate and modify data structures in the form of vectors or multi-dimensional matrices. In that direction, the STATIC and INTERLEAVE schemes are supported. A STATIC distribution causes the memory region being cut in as many chunks as the number of executing threads. Then every chunk is assigned to a core. The INTERLEAVE scheme cuts the memory region in chunks of a particular size, determined at runtime. Then the chunks are assigned to cores in a round robin fashion. Both schemes are totally inspired in their correspondence to loop scheduling schemes. Finally, a RANDOM distribution is supported for generality, to indicate that it is not possible to make a correspondence between the access pattern in the computation and a distribution of a memory region.

The memory regions and their distribution allow for classifying the cache lines and select a specific mechanism to implement the write-back operation. Cache lines are classified as:

- **Volatile** - those that do not belong to any memory region. These cache lines are assumed to be modifiable by any thread at any time and through the High Locality Cache and the Transactional Cache indistinctly.

- **Random** - cache lines that fall only into RANDOM memory region. They are assumed to be modifiable by any thread at any time but exclusively through the High Locality Cache.

- **Speculative** - those that fall either into a STATIC or INTERLEAVE memory region. These cache lines are assumed to be modified by just one thread and exclusively through the High Locality Cache. A particular *owner* concept is associated to these cache lines. Given the base address of a cache line, it is possible to determine the memory region it falls into, and particularly, the core where the cache line belongs to, according to the distribution of the memory region and assignment between chunks of the memory regions and cores. Therefore, the owner of a cache line is the core that owns the chunk of the memory region where the cache line is located.

The aim of the cache line classification is to relax the write-back mechanism. For volatile cache lines, no ownership information is available so a very conservative and costly write-back is designed. For random cache lines, although it is not possible to know what threads are going to modify the cache line, it is ensured that the cache line is exclusively modified through the High Locality Cache. This allows for relaxing the atomicity requirements leading to a more optimized write-back. Finally, speculative cache lines are treated under two assumptions or speculations: first, the cache line is going to be modified under a single-writer scheme; second, a guess is made on the core that is going to modify the cache line. The basis for this speculation is the definition of the owner concept associated to speculative cache lines.

The owner of a cache line is allowed to evict the cache line using a very simple mechanism that does not need to track modified parts nor relies on atomic data transfers. Simply, the

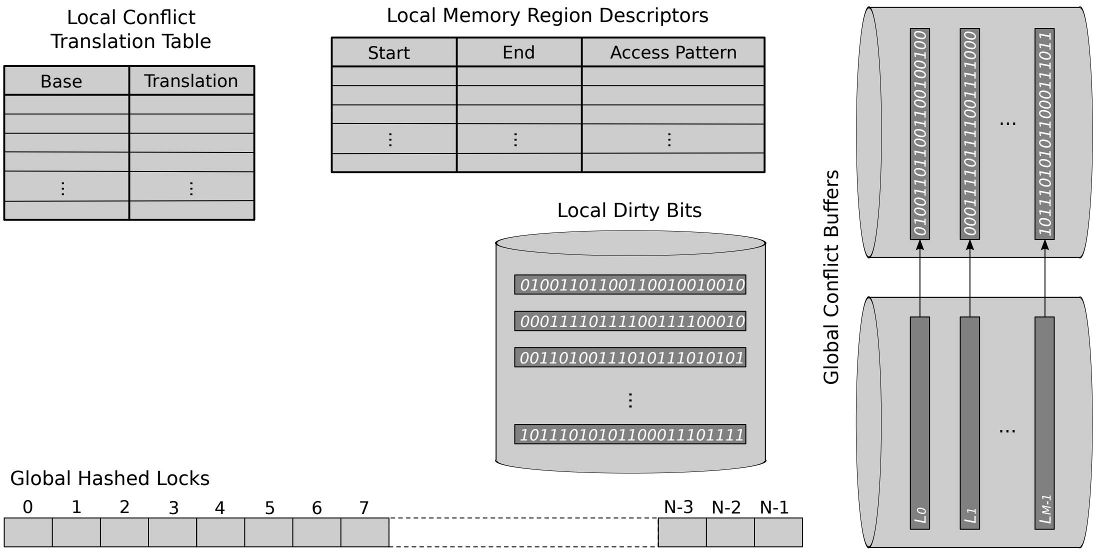


Figure 4.2: Global and local data structures of the enhanced MCB.

owner sends back the modified cache line with no regard of any other write-back operation to be performed by another thread on the same cache line. If a thread evicts a not-owned cache line, this means the speculation was wrong and now the write-back process becomes far more complex, since the consistency protocol is open to the fact that one thread is allowed to freely modify the memory state. In this case, a mandatory synchronization is necessary between the owner and the non-owner, at the moment the latter has to evict a not-owned cache line. Evicting a not-owned cache line is going to require a complex protocol that is likely to result in significant overhead. Although that, notice that the impact on performance strictly depends on how often threads evict not-owned cache lines. The main objective of memory regions is to match the access pattern of the parallel computation, and consequently define appropriate owners. Then the impact of this new approach can be extremely reduced. In order to ensure consistency, owners are guaranteed to be unique and computable by any executing thread. Therefore, the MCB requires that every thread has exactly the same information about memory regions. The task of defining the unique and appropriate memory regions is assigned to the compiler and is discussed in Section 4.3.

Figure 4.2 shows the data structures in the MCB, in particular the memory regions and their distribution are recorded in the Local Memory Region Descriptors. This structure is organized as a table where every entry describes one region and the associated distribution.

Depending on the type of a cache line, a particular write-back mechanism is executed to flush the cache line. Volatile cache lines are evicted using the standard write-back that is already described in Section 3.2.3.1. Random cache lines are evicted using the *Lock-Unlock* write-back (Section 4.2.2) and speculative cache lines are evicted using the *speculative* write-back (Section 4.2.3).

### 4.2.2 Lock-Unlock Write-Back

This write-back mechanism is implemented through atomicity and dirty-bits information, but the effect of atomicity is considerably smoothed. Atomicity is guaranteed through the definition of a mapping between cache lines and a set of locks. The applied mapping function takes the base address of the cache line ($B$) and computes $B$ modulo $N$, where $N$ corresponds to the total number of locks in the MCB. The output of this function points out one of the available locks and it is going to be used at the beginning of the write-back process. Whenever a cache line has to be evicted, the write-back process starts by the locking action over the selected lock. When this operation succeeds, the cache line in main memory is read and transferred to the local memory (this is a temporal metadata introducing additional storage overhead). Then a merge operation is applied between both versions, the one resident in the cache storage and the one recently transferred. Finally, the output of the merge is sent back to main memory and the selected lock is freed. Of course, atomicity is only guaranteed if no thread is allowed to modify the cache line with no other mechanism than the Lock-Unlock write-back. This is ensured by the fact that one line is mapped to only one policy and memory regions are unique among the executing threads (more details follow in Section 4.3). Figure 4.2 shows the runtime structure supporting this mechanism: the Global Hashed Locks structure. It is organized as a vector of $N$ cells, where every one implements a lock data structure.

### 4.2.3 Speculative Write-Back

This write-back mechanism is based on the ownership relation between cores and cache lines. Two different implementations are executed depending on who is executing the write-back process.

The cache line owner is allowed to directly transfer his version of the cache line with no regard of any other ongoing write-back processes operating on the same cache line. Notice that this corresponds to a very efficient write-back, as both atomicity and dirty-bits are avoided.

Non owners postpone the write-back process and temporarily evict the cache line to a buffer allocated in main memory (Figure 4.2, the Global Conflict Buffers). This buffer holds the cache line version plus the dirty-bits content that is going to be necessary when the actual write-back process takes place. Solving the pending write-backs requires isolation in the sense that it is not acceptable to allow one thread to solve a pending write-back while it is possible that another thread freely flushes the same cache line acting as an owner, that is, transfers the cache line with no regard of any other modifying thread. In order to observe this restriction, the execution of pending write-backs is performed by all threads at a time at particular execution points. According to the relaxed memory consistency model, pending write-backs are expected to be solved at a synchronization point located at the end of the parallel computation where the executing threads synchronize under a barrier scheme. Of course, it is possible that the buffers get full in the middle of the parallel execution: then the affected thread blocks the execution until all other threads reach the synchronization point or all threads are blocked because pending write-backs have exceeded their buffer size.

When write-backs are completed, blocked threads resume their execution, and eventually reach the synchronization point. The impact on performance of this process is totally conditioned on how often buffers get full, on the actual write-backs and the cost of the necessary synchronization to start the execution of pending write-backs. Figure 4.2 shows the runtime structures supporting this mechanism. The Global Conflict Buffer corresponds to the buffer that every core uses to postpone the write-back operations. Every core has assigned one exclusive partition of this buffer of $M$ entries. The Local Conflict Translation Table keeps up to $M$ temporary translations for cache lines that have been evicted using the Global Conflict Buffers. In general case, the look-up process of the High Locality Cache is forced to check this structure to maintain the memory consistency. Preferably, parameter $M$ is a small number so as to skip huge overheads for the look-up process of the High Locality Cache. This system is designed to perform with low overheads only when conflicts occur rarely. The Global Barrier Register is used to ensure the isolation along the pending write-back process. The Local Dirty Bits allow for monitoring the modified parts of resident cache lines.

## 4.3 Compiler Support

In this section we describe the compiler technology to activate the capabilities of the cache design at runtime. In particular, memory consistency is addressed and the compiler support to define appropriate memory regions is discussed, as well as the restrictions to be observed in the context of a hybrid design. This section is not intended to describe in full detail a code generation algorithm, but to show that the compiler techniques needed to efficiently exploit the memory consistency mechanisms that have been proposed, are based on well studied techniques, such as array section and pointer analysis.

With no loss of generality, the compiler techniques target the execution of parallel loops, assuming the code transformation that is showed in Figure 4.1(d). The code transformation is performed in four ordered phases:

1. Classification of memory references into high-locality and irregular accesses. The high locality references are those exposing a stride access pattern, all other references are considered irregular;
2. Introducing of memory regions for high-locality memory accesses
3. Code transformation to optimize high-locality memory references
4. Code transformation to optimize irregular memory references.

Compiler support presented in this section corresponds to the phase 2. All other phases are described in Section 3.3.

### 4.3.1 Memory Regions Definition

In the example in Figure 4.1(a), assume the $i$-loop has been parallelized. It is out of the scope of this thesis to cover the mechanisms to support the work distribution among the executing threads. Assume that the iteration space is distributed among the threads according to an OpenMP STATIC scheduling: the $i$-iteration space is cut in even chunks of consecutive iterations and every chunk is assigned to one thread.

In order to generate appropriate and consistent memory regions we rely on two main compiler analysis: array section and pointer analysis. Array section analysis [81] can be applied to determine the range of every write high-locality reference. Ranges can be described with the triplet notation [100]: starting address, ending address and access stride. Clearly, ranges can be directly translated to memory regions. The work of the compiler starts by deciding if any of the available distributions matches how the memory regions are going to be accessed given the loop scheduling and the range information. In case the range information is not sufficient to determine the correspondence to memory regions, a RANDOM distribution can be defined. Remember that the applied distribution is only used for selecting a particular mechanism at the write-back moment. A distribution that at runtime does not match the access pattern in the computation will lead to poor performance, but in any case it is not opening the design to cause memory inconsistency. The only restriction to be observed is that of the possible interaction between the High Locality Cache and the Transactional Cache. Remember that the Transactional Cache operates under a write-through scheme. Because of this behavior, the consistency model is open to allow threads to freely modify the memory state through the Transactional Cache. This collides with assumptions made along the write-back process for cache lines in the range of defined memory regions. It becomes mandatory to ensure that data mapped to the Transactional Cache is never within the range of any defined memory region. For this particular subject, pointer analysis [96, 117] is necessary to determine whether any irregular access can reference a datum within a memory region obtained after the range analysis. If this is the case, the memory region affected by this type of alias can not be defined and all cache lines in the region will be treated as volatile cache lines and evicted with the standard write-back process. In this particular topic, the compiler always has to be conservative. It is not acceptable to define a memory region as long as it can not be ensured that no irregular access is going to modify any datum in the region.

## 4.4 Evaluation

The evaluation section is divided into three parts. First, we measure the performance impact of the proposed memory consistency optimizations. Second, we study the overall performance and compare with TSC, TSC-TILING and PPC970MP configurations. Finally, we measure the overheads introduced by the software cache when using speculative write-back. The CG, IS, FT, and MG applications are used for the evaluation under methodology described in Section 2.2.

Three different hybrid cache configurations are used. The HYBRID is our initial hybrid design (as described in Section 3.2) with standard write-back which treats every modified cache line as volatile. Second configuration is the HYBRID-lock that considers every modified cache line as random. Finally, the last configuration is the one that relies on memory regions for classifying every modified cache line. We refere to this configuration as

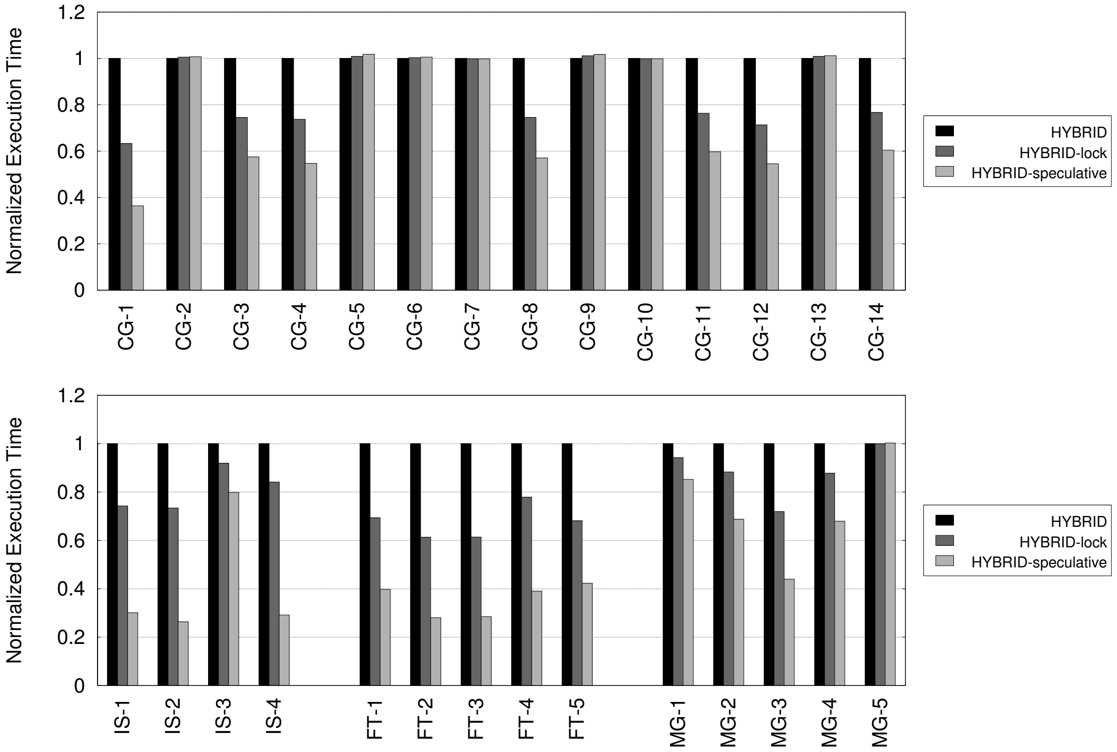


Figure 4.3: Performance impact of memory consistency support in CG, IS, FT, and MG applications.

HYBRID-speculative.

In our implementation for Cell, we use the Global Conflict Buffers of 16 entries per thread, and the Global Hashed Locks structure of 128 locks.

### 4.4.1 Optimized memory consistency effect evaluation

Figure 4.3 shows the performance of every parallel loop (28 loops in total) in the tested applications. The chart shows execution times normalized to the HYBRID configuration. Therefore, the Y-axis shows reduction of the execution time in percentage respect that version.

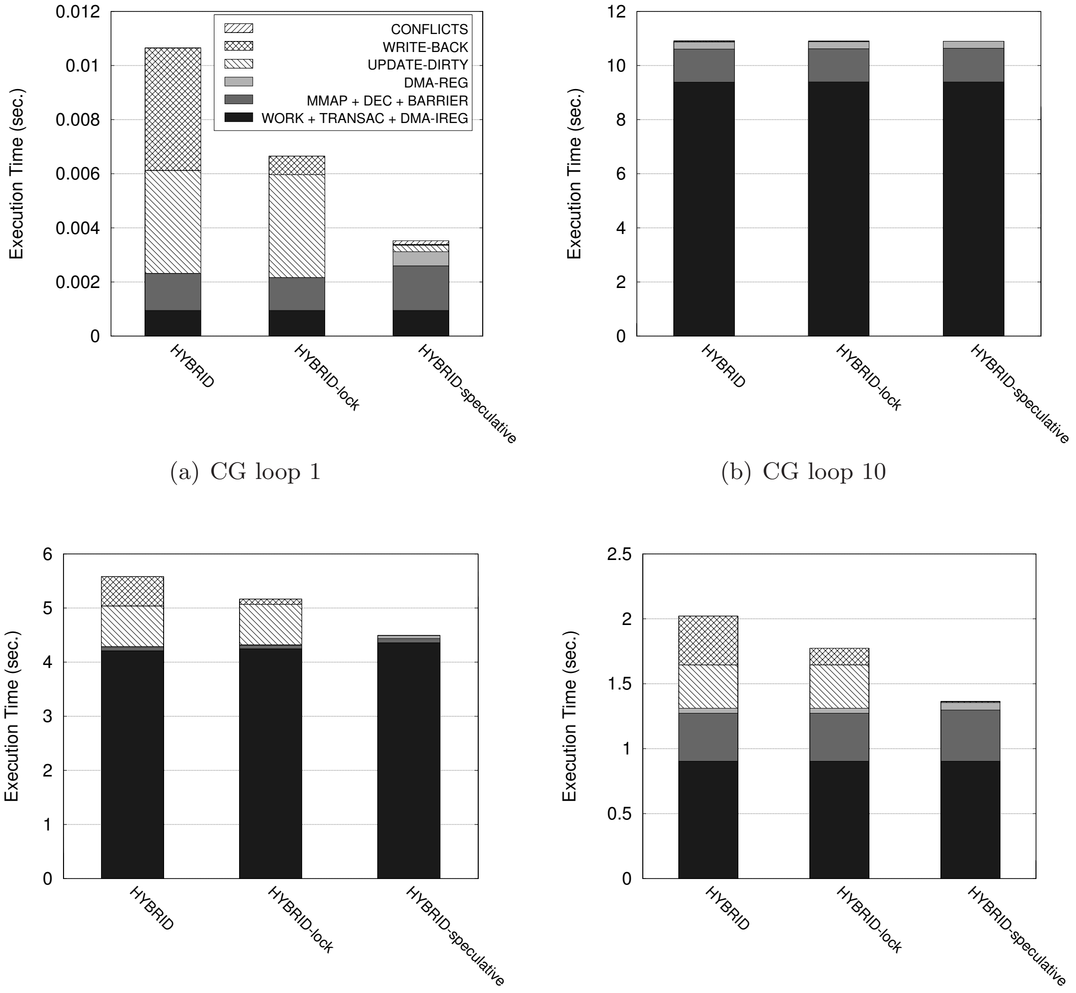
The CG application is composed by 14 parallel loops. Figure 4.3 shows that most of them show a good response in front of the optimizations in the write-back mechanism. This is the case for loops 1, 3, 4, 8, 11, 12, and 14, where the improvements range between 40% and 60% for the HYBRID-speculative versions. Figure 4.4(a) shows the execution time breakdown for loop 1 in CG. Notice how HYBRID-lock version improves only write-back mechanism while update of dirty bits remains the same. In the HYBRID-speculative version both components are improved, being both just 8% of loop execution time, with low overhead imposed by solving pending conflicts. The case of loops 2, 5, 6, 9, and 13 suffer from a reduction operation which modifies only scalar values which makes unnoticeable the improvements in the write-back mechanisms. Finally, loops 10 and 7 suffer from a huge activity in the Transactional Cache. Figure 4.4(b) shows the executing time breakdown for loop 10 in every of its versions. The optimizations in the memory consistency support are unnoticeable in this kind of loops but do not impose any overheads in the control code as well.

The IS application is composed by 4 loops that work with a working set organized under several vectors. Loops 1, 2, and 4 are similar to CG loop 1. These loops show improvements ranging from 20% up to 70%. Overall execution of the IS application is totally dominated by loop 3. This loop suffers from the memory consistency overheads and the activity in the Transactional Cache. Figure 4.4(c) breaks down the execution time of this loop and confirms the expected trend: write-back code is significantly imporoved in the HYBRID-lock version, while both, the update of dirty bits and the write-back code are practically eliminated in the HYBRID-speculative version.

In the MG application the main core of the computation is composed by 5 parallel loops. Unless for loop 5, all loops expose the same trend, in average: the HYBRID-lock version gives an initial improvement about 20%, and the HYBRID-speculative version improves for about 40%. To explain those improvements, Figure 4.4(d) shows the execution time breakdown for loop 4. In the HYBRID version, the write-back and the update of dirty-bits represent 35% of loop execution time, and in the HYBRID-lock only the write-back process is improved, being both issues 28% of total execution time. In the HYBRID-speculative version, both write-back and update of dirty-bits are improved resulting in less than 1% of loop execution time. Solving pending conflicts is negligible in the HYBRID-speculative version of this loop. It is not visibile in Figure 4.4(d) since it corresponds to less than 1% of the execution time. In general, we do not observe any significant overhead for solving conflicts since it was easy to precisely specify memory regions for the tested openmp parallel loops. In general, MG loops 1, 2, 3, and 4 show the same trend, and all of them are affected by a reduced number of conflicts (between 1 and 5, depending on the number of matrices being modified in the loop). This explains the obtained efficiency in the HYBRID-speculative version: very few cache lines are actually evicted relying on dirty bits and atomicity. Finally, loop 5 exposes a very different behavior: this loop computes two reduction operations, thus, it only modifies two scalar values which does not respond in front of the optimized write-back mechanisms.

The FT application is composed by 5 loops that show a behavior very similar to the one exposed by loop 4 in the MG application. Similar results are obtained in what concerns to time distribution. Improvements are about 20-40% for the HYBRID-lock version, and up to 70% for the HYBRID-speculative version.

In conclusion, we observe that the speculative write-back succeeds in avoiding the overheads of the memory consistency support, which ranges from 20% to 80% of loops execution time. In all tested loops the access patterns are easily translated to memory regions and appropriate memory distributions.



(a) CG loop 1

(b) CG loop 10

(c) IS loop 3

(d) MG loop 4

Figure 4.4: Execution time breakdown for CG (loops 1 and 10), MG (loop4) and IS (loop3) applications threated under three different memory consistency mechanisms.

### 4.4.2 Overall Performance

Overall execution times of the tested applications are presented in Figure 4.5. As it is described, MG and FT applications are dominated by high-locality accesses, improvements in those applications are better than in CG and IS which are dominated by transactional accesses. In MG application, HYBRID-lock version gives improvement of 13% and HYBRID-speculative approach gives improvement of 30% versus HYBRID configuration. This is expected since in average all loops in MG show similar improvements. In FT application, reduction in overall execution time of 33% comes from HYBRID-lock version and reduction of 60% comes from HYBRID-speculative version. FT application benefits a lot from HYBRID-speculative version. Obtained execution times for FT are in accordance with

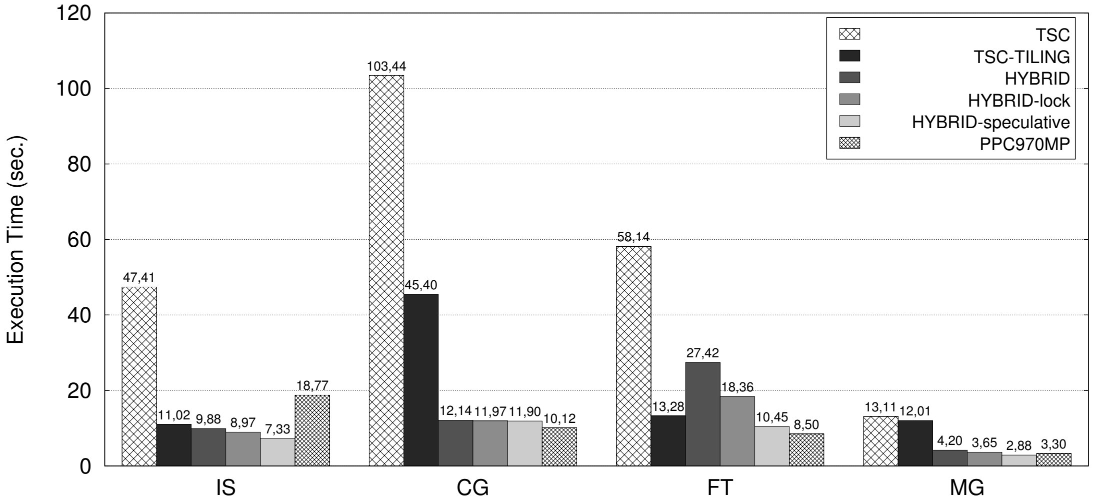


Figure 4.5: Performance of the hybrid software cache approach with memory consistency optimizations.

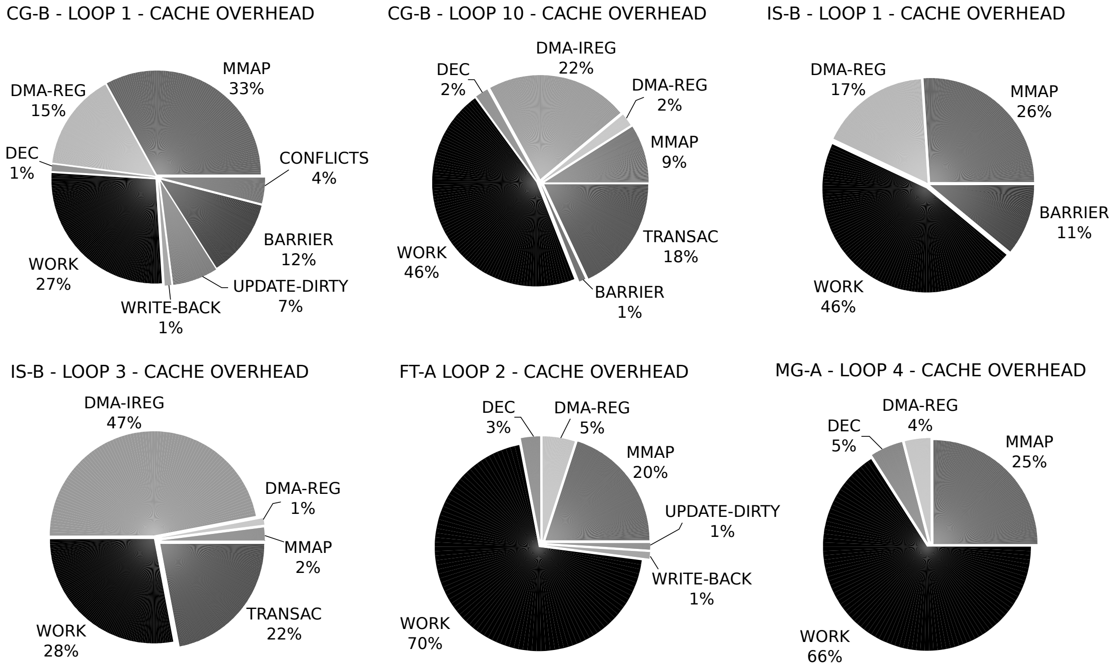
results obtained per loop in FT application. Overall execution time for CG shows that this application does not benefit from memory consistency optimizations. In a per loop analysis, it is shown that some loops benefit from HYBRID-lock and HYBRID-speculative versions in CG but their benefit is not achievable in overall execution time due to loop 10 which is the most consuming loop (90% of execution time) dominated by transactional accesses which does not suffer from memory consistency issues. Besides IS is dominated by transactional accesses, situation is a little bit better than in CG because execution time of the most consuming loop in IS (loop 3) takes 60% of the overall execution time in IS, in contrast to 90% in CG. Improvement of HYBRID-lock version versus HYBRID version in IS is 10% of reduction in overall execution time while improvement from HYBRID-speculative version is around 25%.

Improvement of HYBRID-speculative approach over TSC-TILING is noticeable in all loops, resulting in speedups of 1.50, 3.82, 1.27, 4.17 for IS, CG, FT and MG applications respectively. In respect to PPC970MP configuration, besides having speedup of 2.56 in IS application, HYBRID-speculativeperforms 1.15 times faster in MG application as well. Performance obtained in CG and FT applications for HYBRID-speculative version is near the performance of PPC970MP but still close to 20% behind.

### 4.4.3 Cache Overhead Distribution

The main aim of this section is to analyze the cache overheads and determine an upper bound for the further optimizations. For that purpose, the same six loops from Section 3.4.3 are described here in terms of cache activity under HYBRID-speculative approach. Figure 4.6 shows the time distribution for the selected loops in IS, CG, FT and MG applications.

The case of loop 1 in CG is improved a lot by in the HYBRID-speculative version.



**WORK**: time spent in actual computation.
**UPDATE-DIRTY**: time spent updating the dirty-bits info.
**DMA-REG**: time spent synchronizing the DMA data transfers in the high-locality cache.
**TRANSAC.**: time spent executing control code of the transactional cache.
**MMAP**: time spent in executing lookup, placement actions and DMA programming.
**WRITE-BACK**: time spent in the write back process.
**DMA-IREG**: time spent synchronizing the DMA transfers in the transactional cache.
**DEC.**: time spent in the pinning mechanism for the cache lines.
**BARRIER**: time spent in the synchronization barrier at the end of the parallel computation.
**CONFLICTS**: time spent if solving pending conflicts for memory consistency

Figure 4.6: Cache overhead distribution for CG (loops 1 and 10), IS (loops 1 and 3), FT (loop 2), and MG (loop 4) under HYBRID-speculative configuration. Total execution time is broken down and percentages are displayed. Components of the execution time which correspond to less than 1% of the execution time are omitted from the charts.

Instead of only 9% of loop execution time devoted to computation phase in HYBRID version (Section 3.4.3), now that percentage is 27. It is increased 3 times but still not enough. Due to optimized memory consistency, which now takes only 8% instead of 78% in HYBRID version, a component which is drastically increased now is the synchronization with DMA transfers in the High Locality Cache. This synchronization time takes 15% of loop execution time, while in the HYBRID version it was below 1%. Control code is not long enough anymore to overlap all communication and then the synchronization component becomes noticeable in the HYBRID-speculative configuration. Solving pending conflicts does not incur in unacceptable overheads since 4% of loops execution time is devoted to that, while 12% is devoted to the barrier at the end of parallel section. Barrier overhead is due to small workload granularity. The overhead distribution in CG loop 10 is not changed after memory consistency optimization. It is almost the same as with HYBRID configuration. Some minor changes of up to 1% appear in the control code for High Locality

Cache. In general, this loop is still dominated by 18% of execution in the control code for the Transactional Cache, and by 22% of loop execution in the synchronization with DMA transfers in the Transactional Cache.

The case of loop 1 in IS application corresponds to a loop which benefits a lot from HYBRID-speculative approach. Memory consistency corresponds to less than 1% of loop execution time which makes computation phase to become more significant component, taking 46% of loop execution time now. The trend of increased synchronization time is observed here as well. Synchronizing with DMA transfers in High Locality Cache takes 17% of loop execution, while the rest of time is devoted to the control code in High Locality Cache (26%), and to the synchronization on the barrier at the end of parallel section (11%). Barrier component is also present in this loop because of small workload granularity. The case of loop 3 in IS corresponds to a loop affected by an important amount of activity in the Transactional Cache (22%) and about 47% of the execution is spent in waiting for data transfers in the Transactional Cache. So, even significant synchronization time in the HYBRID version, is more significant now when memory consistency overhead is practically removed.

The cases of loop 2 of FT and loop 4 of MG represents a case which benefits a lot from memory consistency optimization. In FT loop 2, actual computation corresponds to 70% of the total execution time while cache activity is mostly devoted to the mapping of cache lines to the local memory (about 20%). In this loop, only 5% of total time is devoted to synchronization with the DMA transfers in the High Locality Cache. In loop 4 of MG, computation takes 66% of loop execution time, while the rest is mostly devoted to the cache overhead of mapping cache lines to the local memory which takes 25% of execution time.

In general, we do not observe any significant overhead for solving conflicts since it was easy to precisely specify memory regions for the tested openmp parallel loops. The rest of overhead in HYBRID-speculative version is mainly accumulated in synchronizing with the DMA engine, control code of the High Locality Cache and Transactional Cache, the cache line pinning mechanism, and waiting on the barrier. Notice that synchronization with DMA transfers is greater in the HYBRID-speculative than in the HYBRID (Section 3.4.3) configuration since control code is faster and then it is not long enough to overlap with all DMA transfers. In contrast to HYBRID configuration, synchronization with DMA transfers in the High Locality Cache appears to be a problem in HYBRID-speculative configuration.

## 4.5 Related Work

For software maintained memory consistency, there have been several proposals based on adaptive consistency protocols in Software Distributed Shared Memory (SDSM) systems [8, 9, 19, 35, 54]. In order to reduce consistency-related communication, those approaches try to match the data flow in the computation to several schemes that describe how the data is accessed and modified. The single writer and multiple writer access patterns configure the consistency protocol. Besides, the adaptability of the consistency protocols also targets the minimization of the communication events, by dynamic page aggregation. Basically, the runtime responsible for all of the communication is able of coalescing data transfers affected by the same consistency attributes. In these works, the compiler is not involved in providing any type of information to guide the system adaptability.

Therefore, Dwarkadas et al. [36] proposed an integrated compile-time/runtime SDSM system. This integrated system is based on explicit compiler-generated calls for performing aggregated communication prior to loop executions in order to minimize costly runtime interventions. Regular section descriptors are used to identify opportunities for communication optimizations which resulted in a good performance for regular applications. Later, Lu et al. [65] extended this integrated SDSM system with a compiler support for irregular applications. Basically, compiler generates data access information for irregular access patterns prior to loop executions, enabling a runtime system to precompute the set of pages that will be accessed by each core through irregular accesses. Then, the runtime system requests precomputed pages prior to the loop execution, reducing the amount of communication, if any, needed during the loop execution itself. Efficiency of this approach greatly depends on how precise compiler analysis can be for the irregular access patterns.

Many differences arise between these works and the proposal in this chapter. First, the presented mechanisms are adaptive and speculative in the sense that the implementation works with non exact information about whether or not a cache line is going to be modified by one or more threads. Adapting the consistency protocol according to single writer and multiple writer schemes requires precise information about the data that is going to be modified in each execution flow. Moreover, in the integrated SDSM systems, compiler has to generate precise information about data accesses in order to achieve efficient communication. This requires a complex compiler analysis. In our system, compiler provides with hints to configure the runtime mechanisms supporting the memory consistency for regular access patterns. For irregular access patterns, we do not use any compiler analysis to predict data accesses, instead we increase communication and computation overlap as addressed in Chapter 5. Finally, in SDSM systems data is distributed and challenges are imposed on reducing the amount of communication to remote nodes, while in our system data is centralized in the main memory, since local memories are to small to contain the whole applications' workload. So, SDSM systems have to maintain memory consistency across nodes, while in our work we have to maintain the consistency of the main memory. However, one similarity is imposed. When false sharing happens in our system we use less efficient write-back mechanism that is based on merging the modified parts of the cache line to the original cache line, according to the dirty bits information. This is similar in the SDSM systems, since merging the modified parts enables communication reduction by transferring only modified data through the network. A structure similar to the dirty-bits is used to track the modifications in the SDSM systems as well.

Costa et al. [29] studied a runtime alignment of loop boundaries to page boundaries. This technique collects runtime information about page faults in SDSM systems. Data structures are build to describe which thread is modifying what page and in what particular iteration. We observe two different aspects: first, the information collected at runtime does not relate loop iterations with cache lines, and secondly, speculative techniques are introduced guided by hints introduced by the compiler.

## 4.6 Conclusions

This chapter describes novel memory consistency mechanisms in the context of software cache techniques. The main achievement is that of minimizing the impact on performance for the most critical aspects in software-based implementations: atomicity and control code execution related to dirty-bits data structures. Performance evaluation indicates that the proposed mechanisms are significantly more efficient: speedup factors between 20% and 40% are observed in several applications in the scientific domain.

Overhead distribution analysis for the HYBRID-speculative configuration confirms that overhead of synchronizing with DMA transfers in both High Locality Cache and Transactional Cache is a candidate to be optimized. The potential effect of overlapping the communication with computation ranges between 1% and 40%, depending on the loop type. For loops dominated by activity in the High Locality Cache, up to 20% of the loop execution time is spend in the non-overlapped communication. Loops dominated by activity in the Transactional Cache might experience significant improvements of close to 40%. Besides the potential positive effect of prefetching we have to point out, as a negative aspect of the software cache, that the cache overhead of mapping data to the local memory (control code of High Locality Cache and Transactional Cache) is a significant chunk of execution time, sometimes more than 30% of loop execution time. In the next chapter, we address overlapping of the communication with computation in order to decrease the overhead of synchronizing with DMA transfers. Also, the essential overhead of mapping data to the local memory is addressed by some hardware optimizations in Chapter 7.

# Chapter 5

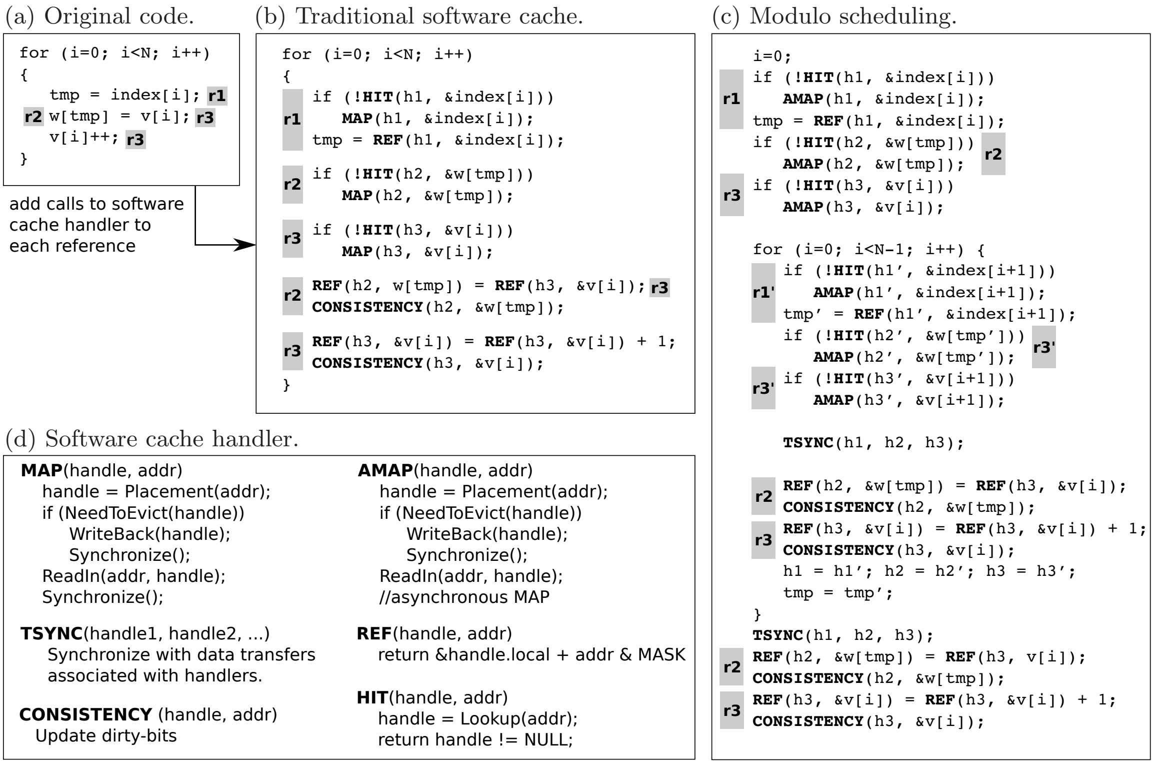
# Prefetching

This chapter is about prefetching techniques used to achieve efficient data transfers by increasing the overlap of communication and computation in our hybrid access-specific software cache. This chapter is organized as follows. Motivation of the proposal is presented in Section 5.1. The detailed design of the prefetching support for the High Locality Cache and the Transactional Cache is presented in Section 5.2, while Section 5.3 explaines needed compiler transformation. Section 5.4 evaluates our proposal. Related work is presented in Section 5.5 and conclusions of the chapter are in Section 5.6.

## 5.1 Motivation

Smoothing the impact of the synchronous DMA transfers in order to achieve good overlap of the communication with the control code execution is significantly addressed in our hybrid software cache. In contrast to traditional software cache, all DMA transfers that appear in our hybrid design are scheduled on a way to be overlapped at least with some control code execution. Overlap of DMA transfers which are bringing data is supported through grouped control code and asynchronous DMA transfers, while overlap of DMA transfers which are moving modified data to main memory is supported by eager write-back in the High Locality Cache, and by write-through policy in a combination with partitioning scheme in the Transactional Cache. As described in Chapter 3, this communication overlap is one of the main reasons for great improvements in respect to traditional software cache. According to the analysis in Section 3.4.2, especially good overlap is achieved in the Transactional Cache.

However, analysis of the cache overheads in the previous chapter (Section 3.4) shows that synchronizing with DMA transfers is still an important overhead. Whatever the implementation of the control code for bringing data to local memory, it is necessary to introduce a synchronization between the data transfers and the actual load/store operation the control code is associated to. This typical scheme of control code followed by the synchronization and the associated computation phase (*control-synch-compute* scheme), hinders the possibility of overlapping communication with computation in both cache geometries of our hybrid design. Prefetch techniques can be introduced to hide the memory latencies, but in



(a) Original code.

```
for (i=0; i<N; i++)
{
   tmp = index[i];
   w[tmp] = v[i];
   v[i]++;
}
```

(b) Traditional software cache.

```
for (i=0; i<N; i++)
{
   if (!HIT(h1, &index[i]))
      MAP(h1, &index[i]);
   tmp = REF(h1, &index[i]);

   if (!HIT(h2, &w[tmp]))
      MAP(h2, &w[tmp]);

   if (!HIT(h3, &v[i]))
      MAP(h3, &v[i]);

   REF(h2, w[tmp]) = REF(h3, &v[i]);
   CONSISTENCY(h2, &w[tmp]);

   REF(h3, &v[i]) = REF(h3, &v[i]) + 1;
   CONSISTENCY(h3, &v[i]);
}
```

(c) Modulo scheduling.

```
i=0;
if (!HIT(h1, &index[i]))
   AMAP(h1, &index[i]);
tmp = REF(h1, &index[i]);
if (!HIT(h2, &w[tmp]))
   AMAP(h2, &w[tmp]);
if (!HIT(h3, &v[i]))
   AMAP(h3, &v[i]);

for (i=0; i<N-1; i++) {
   if (!HIT(h1', &index[i+1]))
      AMAP(h1', &index[i+1]);
   tmp' = REF(h1', &index[i+1]);
   if (!HIT(h2', &w[tmp']))
      AMAP(h2', &w[tmp']);
   if (!HIT(h3', &v[i+1]))
      AMAP(h3', &v[i+1]);

   TSYNC(h1, h2, h3);

   REF(h2, &w[tmp]) = REF(h3, &v[i]);
   CONSISTENCY(h2, &w[tmp]);
   REF(h3, &v[i]) = REF(h3, &v[i]) + 1;
   CONSISTENCY(h3, &v[i]);
   h1 = h1'; h2 = h2'; h3 = h3';
   tmp = tmp';
}
TSYNC(h1, h2, h3);
REF(h2, &w[tmp]) = REF(h3, v[i]);
CONSISTENCY(h2, &w[tmp]);
REF(h3, &v[i]) = REF(h3, &v[i]) + 1;
CONSISTENCY(h3, &v[i]);
```

(d) Software cache handler.

```
MAP(handle, addr)
   handle = Placement(addr);
   if (NeedToEvict(handle))
      WriteBack(handle);
      Synchronize();
   ReadIn(addr, handle);
   Synchronize();

TSYNC(handle1, handle2, ...)
   Synchronize with data transfers
   associated with handlers.

CONSISTENCY (handle, addr)
   Update dirty-bits
```

```
AMAP(handle, addr)
   handle = Placement(addr);
   if (NeedToEvict(handle))
      WriteBack(handle);
      Synchronize();
   ReadIn(addr, handle);
   //asynchronous MAP

REF(handle, addr)
   return &handle.local + addr & MASK

HIT(handle, addr)
   handle = Lookup(addr);
   return handle != NULL;
```

Figure 5.1: Modulo scheduling transformation as a prefetching technique.

the context of software cache systems, prefetch does not come for free. Prefetching requires execution of control code related to the look-up, placement/replacement and data transfer operations. Besides, it is necessary to ensure that the prefetched data is in the range of the valid address space.

One well known concept to break down the typical control-synch-compute scheme is shown in Figure 5.1. In Figure 5.1(b), a code transformation for the traditional software cache is applied for a code example in Figure 5.1(a). A code transformation for the traditional software cache uses MAP handler to get data and wait until it comes. In order to enable overlap of the communication and computation it is necessary to break synchronous communication in the MAP handler into calls: asynchronous MAP (AMAP) to get data, and TSYNC to wait until the data has arrived (descriptions of MAP, AMAP, TSYNC and other handlers are shown in Figure 5.1(d)). Now, it is possible to schedule a work between a given AMAP and its associated TSYNC in order to overlap communication with computation. Modulo scheduling execution [61, 87, 88] in one such technique used to schedule a work in between AMAP and TSYNC. In Figure 5.1(c), we illustrate what changes should be made in order to apply this technique to a code operating with traditional software cache. Basically, data used in iteration *i+1* is prefetched in iteration *i*. Asynchronous communications in AMAP makes possible to overlap some computation in iteration *i* with some communication related to data used in iteration *i+1*. Notice that

TSYNC call, is not associated to the last executed control code burst as it would be in the typical control-synch-compute scheme. TSYNC call in iteration *i+1* is associated to the control code executed in iteration *i*. However, the modulo scheduling transformation is not easy to support, especially in the traditional software cache. There are two undesirable situations that make the transformation in Figure 5.1(c) inapplicable:

- It is necessary to ensure that no conflicts appear between the set of consecutive AMAP operations. This is related to the associative level of the cache design and suggests a full associative scheme, always limited by the look-up overheads.
- It is possible that some write-back operation is triggered along an AMAP operation. If it implies some communication that has to be performed synchronously, then the effect of the modulo scheduling transformation is useless.

In contrast to traditional software cache, our hybrid design does not suffer from the mentioned undesirable situations. Fully associativity scheme is already required by our code transformation for both cache geometries due to grouped control code execution which is already present in our design. Also, synchronous transfers are highly avoided in our design due to write-through policy in the Transactional Cache, and due to speculative and adaptive memory consistency support for the High Locality Cache.

Thus, our proposal is to improve our hybrid cache design to include specific support of overlapping communication and computation. According to the nature of memory accesses and in order to exploit performance, our support of overlapping communication and computation is access-specific. We use specific support for automatic prefetch in High Locality Cache, and we adapt the design of the Transactional Cache to efficiently support modulo scheduling transformation.

Performance evaluation indicates that optimized structures of our hybrid software cache combined with the proposed prefetch techniques translate into speed-up between 10% and 20% for a set of parallel NAS applications. Finally, as a result of all proposed optimizations for software caching, we can achieve similar performance on the Cell BE processor as on a modern server-class multi-core such as the IBM PowerPC 970MP processor.

## 5.2 Modified Software Cache Design

In this section, we describe our software cache design improved to enable automatic prefetch for regular memory references and to enable modulo scheduling transformations for irregular memory references. Software cache design described in Chapter 3 is extended with one additional block named Prefetch Block in Figure 5.2.

### 5.2.1 The High Locality Cache

The High Locality Cache structures and operational model are the same as it is described in Section 3.2.1 except that the operational model is extended with the functionality to communicate with Prefetch Block. The High Locality Cache can communicate with the Prefetch Block in order to trigger prefetching of a cache line and during that communication

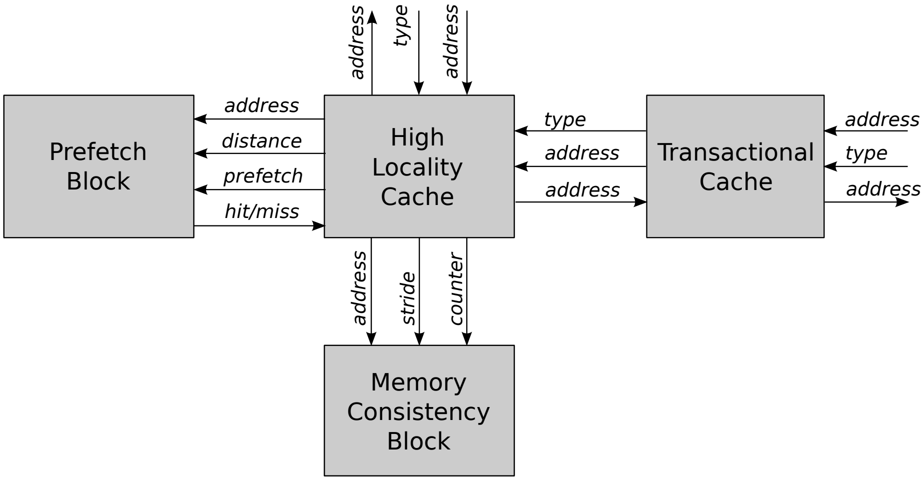


Figure 5.2: Block diagram of our improved software cache design.

two parameters have to be passed to the prefetch block: the global base address of the cache line that triggers prefetch and the distance, in a number of cache lines, from the cache line that triggers prefetch to the cache line to be prefetched. This is everything about new features in the High Locality Cache.

### 5.2.2 The Prefetch Block

The Prefetch Block enables automatic prefetch for regular memory references. The Prefetch Block is selective in the sense that not all regular memory references trigger the prefetch. It is activated under demand according to the activity in the High Locality Cache. For selected references, the memory addresses are forwarded to the Prefetch Block. Then the prefetch can be activated and all forwarded addresses determine the next cache lines to be prefetched.

#### 5.2.2.1 The Prefetch Structures

The Prefetch Block is composed of the following four structures:

- Prefetch Translation Record to preserve for each reference the address resolved by the prefetch operation
- Prefetch Translation Table to keep track of records being used in prefetch operation
- The Prefetch Communication Tags to preserve DMA tags used for prefetching.

The Prefetch Translation Record structure consists of four fields:

- The prefetch global address is the base address of the cache line that triggers prefetch
- The prefetch local address is the base address of the cache line allocated to hold the prefetched data in the local memory

- the prefetch cache line descriptor is a pointer to the cache line descriptor of the prefetched line

- The prefetch distance that indicates the next cache line to be prefetched as a distance (in a number of cache lines) from the cache line base address that triggered the prefetch.

The Prefetch Translation Table is a table where each entry holds one Prefetch Translation Record. The Prefetch Counter keeps track of the number of pending prefetch operations.

The Prefetch Communication Tags consists of all communication tags actively used for prefetching purposes. These tags are going to be used to synchronize the data transfers associated to the prefetched data.

#### 5.2.2.2 The Prefetch Block Operational Model

Memory references that have been selected to trigger the prefetch are recorded in the Prefetch Translation Table. Prefetch is activated from the High Locality Cache and this causes the Prefetch Block to traverse the Prefetch Translation Table and for every non empty entry performs the look-up, placement and replacement operations as if the cache line being prefetched was referenced by the actual computation. Along this process all the communication tags used in the data transfers are recorded in the Prefetch Communication Tags register. Under control of the High Locality Cache, it is possible to synchronize with the prefetched data using this register.

Introducing prefetch support requires reserving some of the available communication tags specifically for that purpose. The range of tags that was used to bring data in to the cache storage is split in two different ranges (in our implementation for Cell, one range is from 0 to 7, and the other from 8 to 15). Both ranges are assigned in a circular manner and the High Locality Cache and the Prefetch block are coordinated to switch from one range to the other every time the Prefetch block is required to perform prefetch operations.

### 5.2.3 The Transactional Cache

Transactional Cache structures and the operational model described in the Section 3.2.2 are changed in order to support efficient modulo scheduling transformations.

#### 5.2.3.1 The Transactional Cache Structures

Essentially the Transactional Cache structures are untouched. The major changes appear in partitioning of the directory and storage structures. To increase concurrency, the cache directory and storage structures are logically divided in four equal-size partitions (Figure 5.3). The Cache Turn Ticket indicates which partition is actively used. Within the active partition, the Cache Placement Index points to the cache line that will be used to service the next miss.

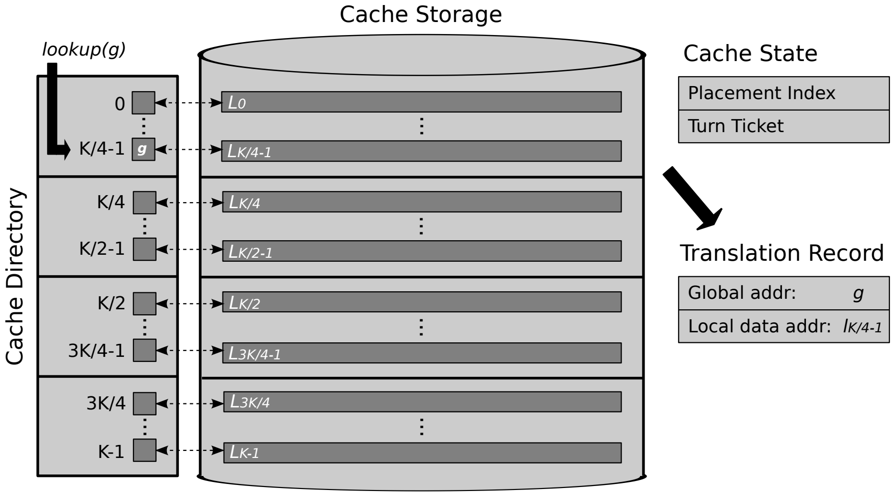


Figure 5.3: Structures of the Transactional Cache adapted for modulo scheduling.

At a high level, the active partition is used for buffering cache lines which are going to be used in the current transaction and these cache lines were prefetched. The next partition, in circular manner, is used for placing cache lines which we are prefetching and which are going to be used in the next transaction in the next iteration of the unrolled loop. Other two partitions are used to buffer data of the two previous transactions while their modified data is being flushed back to the main memory. Based on this explanation, we defined three states in which our partitions can be: *in-use*, *prefetching* and *flushing*.

#### 5.2.3.2 The Transactional Cache Operational Model

A transaction has the same meaning as it is explained in Section 3.2.2.2 and operations in a transaction happen, as usual, in four consecutive steps: (1) initialization, (2) communication into local memory, (3) computation associated with the transaction, and (4) propagation of any modified state back to global memory. All changes in operational model are done due to new partitioning scheme.

In addition to the explanations about Step 1 (Section 3.2.2.2), during this step the Cache Placement Index can have values 0, $K/4$, $K/2$, and $3K/4$ (in our implementation for Cell, these values are 0, 8, 16 and 24 respectively) when the Cache Turn Ticket is, respectively, pointing to partition 0, 1, 2, and 3.

The major change is done in Step 2 in the look-up, placement and replacement routines. Just to remember, the look-up process for a given reference $r$, translation record $h$ and global address $g$ proceeds first with look-up in High Locality Cache and if miss occurs then it proceeds with look-up in a Transactional Cache directory. If miss occurs in the Transactional Cache directory system operates as usual (Section 3.2.2.2). Handling of the hit within the Transactional Cache directory is now different. When a hit occurs, the look-up can operate in one of two ways. If the hit occurred within the active partition (partition where we are going to prefetch the data for next iteration), we simply update the translation record *h*. If the hit occurred within the next partition, in circular manner, then we need to do two actions. First, we need to migrate the line to the active partition, a placement operation is used for this operation as well. Second, we need to inform previous partition (partition which is in in-use state) about migrated cache line in order to maintain consistency between transactions. If however hit occurs within the other partitions, we simply update the translation record *h*.

In the synchronization routines that are used to ensure consistency within and across transactions, we have to be aware of using exact tags for synchronization which are not the same as in the Section 3.2.2.2. When partition 0 is active, we wait for data transfer operations with tags $[0 \ldots \frac{K}{4} - 1]$, for partition 1 appropriate tags are $[\frac{K}{4} \ldots \frac{K}{2} - 1]$, for partition 2 tags are $[\frac{K}{2} \ldots \frac{3K}{4} - 1]$, and for partition 3 wait is for tags $[\frac{3K}{4} \ldots K - 1]$. Note that in our implementation for Cell, these groups of tags are $[0 \ldots 7]$, $[8 \ldots 15]$, $[16 \ldots 23]$, and $[24 \ldots 31]$ respectively for partitions 0, 1, 2, and 3.

## 5.3 Code Transformations

We describe in this section the type of code transformation techniques that are now enabled using our prefetching and modulo scheduling approach in the software cache. With no loss of generality, the code transformation targets the execution of loops.

The code transformations are performed as usual (Section 3.3) in three ordered phases:

1. Classifying of memory references into regular and irregular accesses
2. Transformation of the code to optimize regular memory accesses
3. Transformation of the code to optimize irregular memory accesses.

This process is illustrated in Figure 5.4.

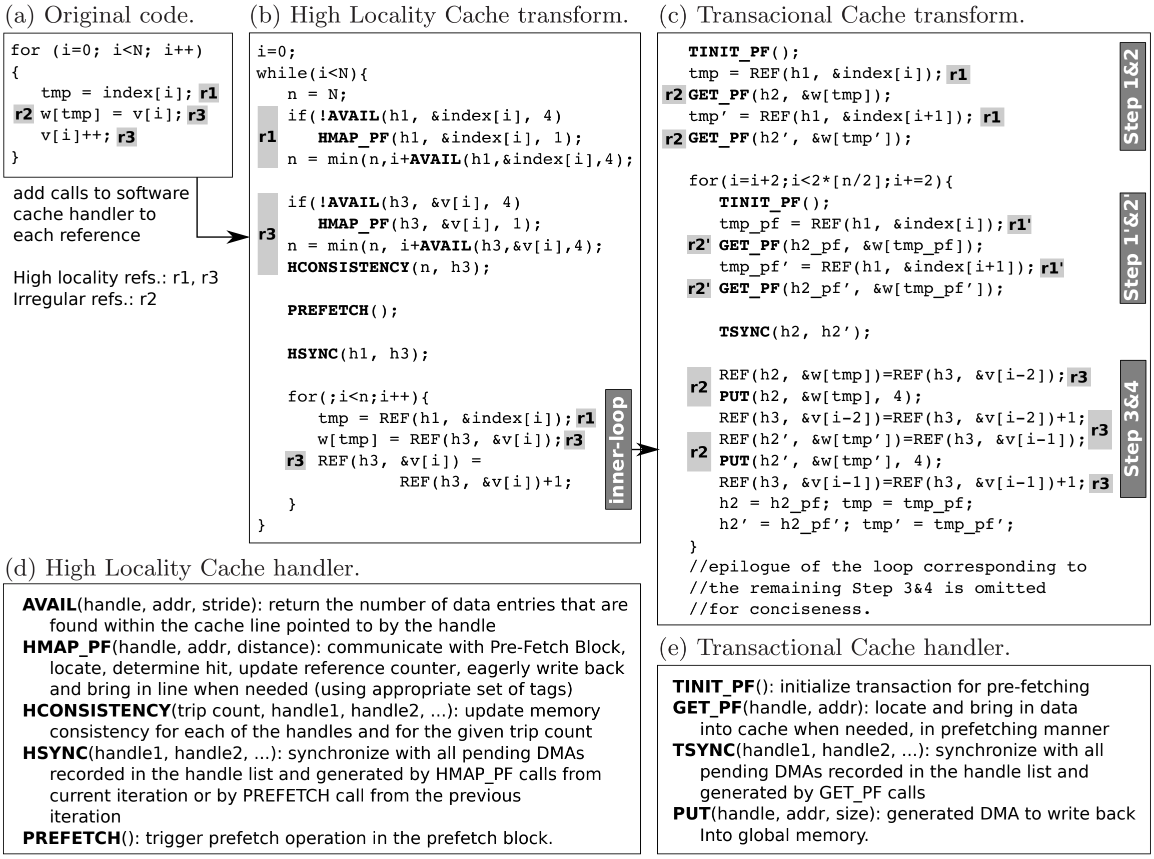
First phase is the same as it is described in Section 3.3, references *r1* and *r3* are regular and reference *r2* is irregular (Figure 5.4(a)). Second and third phase are different and here we are going to explane those differences.

### 5.3.1 High-Locality Access Transformations

Code transformation related to the regular references is not significantly changed. There are two major changes: new handler named HMAP_PF is used instead of old HMAP handler and in addition to prior transformations, PREFETCH handler is provided for prefetching purposes.

Just to remember, original *for*-loop is transformed into two nested loops (Figure 5.4(b)). Work done in the inner *for*-loop (related to regular memory accesses) does not have any cache overhead. In the while loop we are introducing necessary code transformations per each regular memory reference. The look-up, dynamic sub-chunking, consistency maintaining, prefetching and synchronization operation are done in the while loop.

The look-up operation checks if the address *&index[i]* of the reference *r1* is in the cache line pointed to by the translation record (handle) *h1*. This checking is done by



(a) Original code.

```
for (i=0; i<N; i++)
{
   tmp = index[i]; r1
r2 w[tmp] = v[i]; r3
   v[i]++; r3
}
```

add calls to software cache handler to each reference

High locality refs.: r1, r3
Irregular refs.: r2

(b) High Locality Cache transform.

```
i=0;
while(i<N){
   n = N;
   if(!AVAIL(h1, &index[i], 4)
r1    HMAP_PF(h1, &index[i], 1);
   n = min(n,i+AVAIL(h1,&index[i],4);

   if(!AVAIL(h3, &v[i], 4)
      HMAP_PF(h3, &v[i], 1);
r3 n = min(n, i+AVAIL(h3,&v[i],4);
   HCONSISTENCY(n, h3);

   PREFETCH();

   HSYNC(h1, h3);

   for(;i<n;i++){
      tmp = REF(h1, &index[i]); r1
      w[tmp] = REF(h3, &v[i]); r3
   r3 REF(h3, &v[i]) =
               REF(h3, &v[i])+1;
   }
}
```

(c) Transacional Cache transform.

```
   TINIT_PF();
   tmp = REF(h1, &index[i]); r1
r2 GET_PF(h2, &w[tmp]);
   tmp' = REF(h1, &index[i+1]); r1
r2 GET_PF(h2', &w[tmp']);

   for(i=i+2;i<2*[n/2];i+=2){
      TINIT_PF();
      tmp_pf = REF(h1, &index[i]); r1'
   r2' GET_PF(h2_pf, &w[tmp_pf]);
      tmp_pf' = REF(h1, &index[i+1]); r1'
   r2' GET_PF(h2_pf', &w[tmp_pf']);

      TSYNC(h2, h2');

   r2 REF(h2, &w[tmp])=REF(h3, &v[i-2]); r3
      PUT(h2, &w[tmp], 4);
      REF(h3, &v[i-2])=REF(h3, &v[i-2])+1;
   r2 REF(h2', &w[tmp'])=REF(h3, &v[i-1]); r3
      PUT(h2', &w[tmp'], 4);
      REF(h3, &v[i-1])=REF(h3, &v[i-1])+1; r3
      h2 = h2_pf; tmp = tmp_pf;
      h2' = h2_pf'; tmp' = tmp_pf';
   }
   //epilogue of the loop corresponding to
   //the remaining Step 3&4 is omitted
   //for conciseness.
```

(d) High Locality Cache handler.

**AVAIL**(handle, addr, stride): return the number of data entries that are found within the cache line pointed to by the handle
**HMAP_PF**(handle, addr, distance): communicate with Pre-Fetch Block, locate, determine hit, update reference counter, eagerly write back and bring in line when needed (using appropriate set of tags)
**HCONSISTENCY**(trip count, handle1, handle2, ...): update memory consistency for each of the handles and for the given trip count
**HSYNC**(handle1, handle2, ...): synchronize with all pending DMAs recorded in the handle list and generated by HMAP_PF calls from current iteration or by PREFETCH call from the previous iteration
**PREFETCH**(): trigger prefetch operation in the prefetch block.

(e) Transactional Cache handler.

**TINIT_PF**(): initialize transaction for pre-fetching
**GET_PF**(handle, addr): locate and bring in data into cache when needed, in prefetching manner
**TSYNC**(handle1, handle2, ...): synchronize with all pending DMAs recorded in the handle list and generated by GET_PF calls
**PUT**(handle, addr, size): generated DMA to write back Into global memory.

Figure 5.4: Example C code and its prefetching and modulo scheduling code transformation. The assumed size of the data found in the used arrays is 4 bytes.

using AVAIL macro. The AVAIL macro returns for reference *r1* number of iterations for which this reference will be present in the cache line pointed to by handle *h1*. If this number is greater than zero we have hit and then we are proceeding with determining of the upcoming dynamic sub-chunk of the iteration space. If this number is equal to zero then macro HMAP_PF is invoked to serve a miss. Notice the third argument of the HMAP_PF macro, indicating if prefetch has to be considered for the given memory reference. This argument corresponds to the prefetch distance, indicating the next cache line to be most likely accessed by the memory reference. In case the distance is other than zero, prefetch is activated and the address is forwarded to the Prefetch Block. Next step is determining of the upcoming dynamic sub-chunk of the iteration space. Once we have sub-chunking factor *n* we can process with consistency and synchronization operations. Since reference *r1* is read only access reference then consistency operation is not processed for *r1* but is processed for *r3* which is read and write access reference. The PREFETCH macro triggers prefetch for all forwarded addresses. Notice that the prefetch code is executed before the synchronization with the DMA engine takes place, giving the opportunity to overlap the

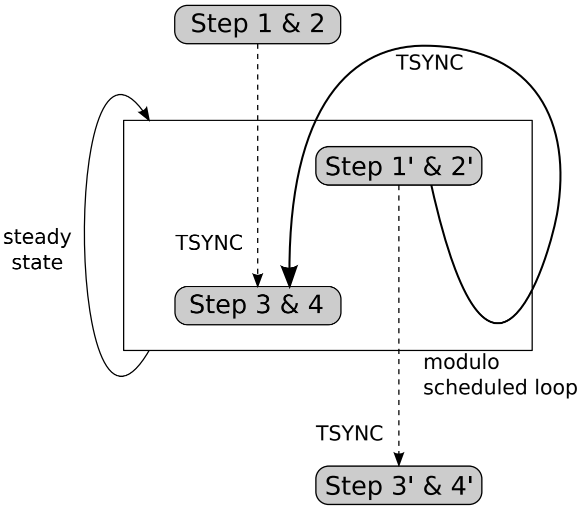


Figure 5.5: Applied modulo scheduling.

prefetch code to actual communication.

#### 5.3.1.1 Irregular Access Transformations

All work done in Section 3.3.3 is valid here and in addition to that work, we apply modulo scheduling transformation exploiting advantage of four-partitioned Transactional Cache. Transaction identification and code generated for a transaction follow the same concept described in initial code transformation (Section 3.3.3). In addition to initial code transformation, we apply modulo scheduling and we cross transactions among loop iterations.

Conceptually, the work inside transactions, in modulo scheduled loop, can be visualized as four tasks. This is shown in Figure 5.4(c). In the loop prologue we prefetch data which are going to be used within computation section in the first iteration of the unrolled loop. In the prologue we use translation records *h2* and *h2'*. We assign task *Step1&2* to the prologue. At the beginning of the unrolled loop body we prefetch data which is going to be used in the next iteration or in the loop epilogue. In this part of the code we use translation records *h2_pf* and *h2_pf'*. We assign task *Step1'&2'* to this part of the unrolled loop body. After this we have a necessary synchronization point where we synchronize with pending DMAs determined by translation records *h2* and *h2'*. When we are sure that data has arrived in the cache, we execute computation section and at the end, modified data is sent back to main memory (PUT macro). This corresponds to task *Step3&4*. In the end of the unrolled loop body translation records *h2* and *h2'* are updated with values from translation record *h2_pf* and *h2_pf'* respectively. In the steady state of the loop, partitions go changing of state: *prefetch*, *in-use*, and *flushing*. Note that for conciseness, the loop unrolling has been done assuming that the number of iterations was a multiple of two. Fourth step named *Step3'&4'* is placed in the epilogue of the loop. Epilogue is emitted here for conciseness. Applied modulo scheduling is illustrated in Figure 5.5.

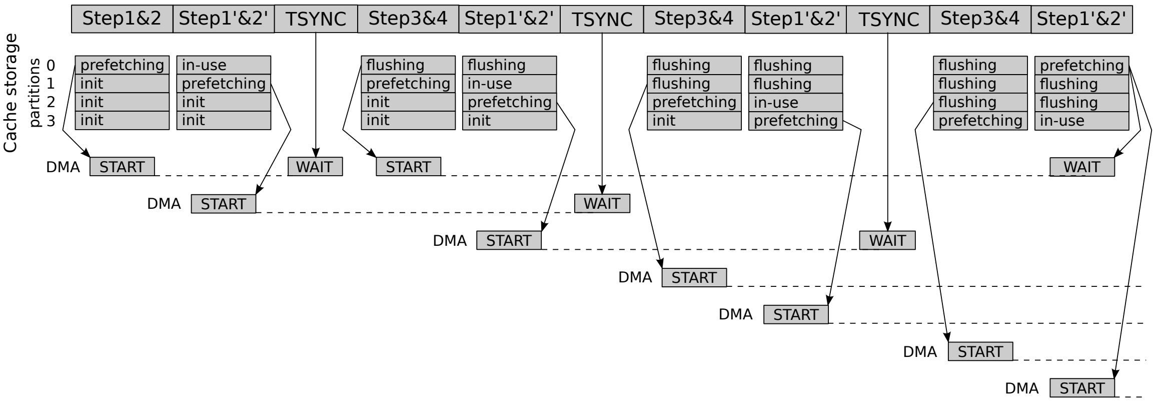


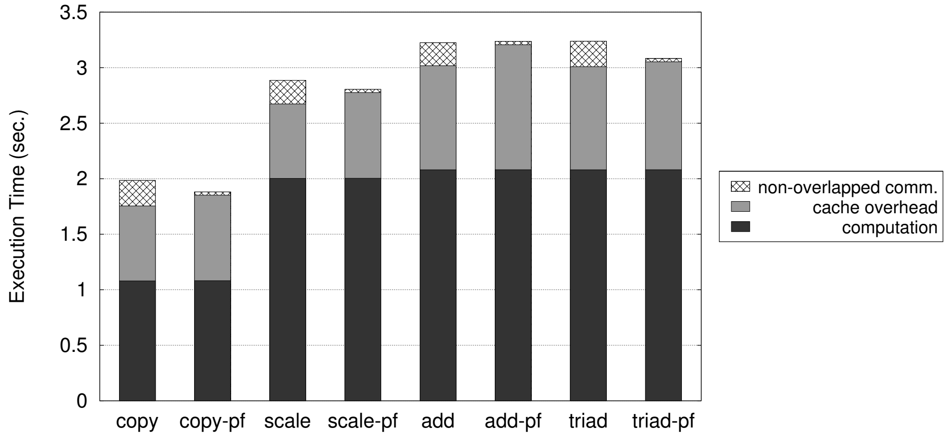
Figure 5.6: Sequence of events in a modulo scheduled loop.

Figure 5.6 shows the evolution of each partition for three iterations of the loop. Distance between starting of DMA and synchronizing on TSYNC point is one iteration of the unrolled loop. Except in first iteration where TSYNC synchronizes with data prefetched in the loop prologue. Synchronization point inside *Step1&2* is placed two iterations far from starting of appropriate DMA commands what is very important because in this point we synchronize with data being flushed back. Hence only modified parts (less than 16 bytes) of the cache lines (not whole cache lines) are flushed back, it means that DMA is programmed to work with inefficient transfer sizes. Mentioned distance of two iterations gives us opportunity for good overlap of computation and communication.

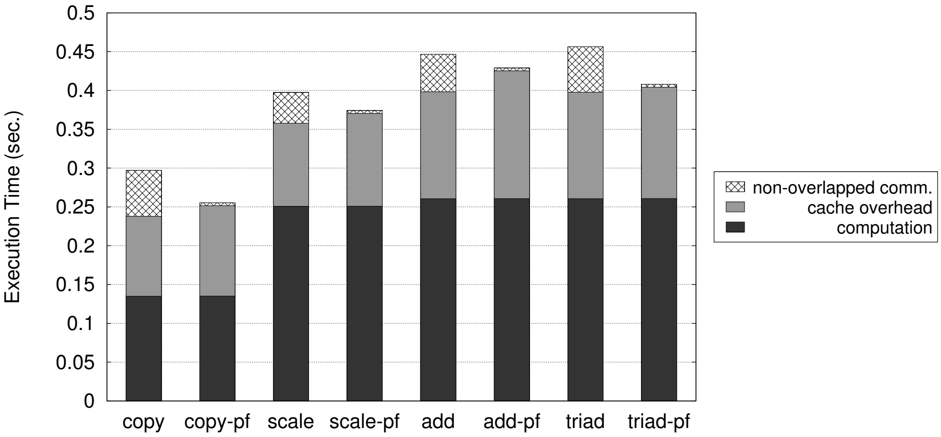
## 5.4 Evaluation

The evaluation section is divided into two parts. First, we measure the impact in performance of the proposed prefetching techniques: automatic prefetch for regular references, modulo scheduling for irregular references. Second, we study the overall performance and compare with TSC, TSC-TILING, and PPC970MP configurations. For the prefetch effect evaluation, the STREAM and the RandomAccess synthetic kernels are used, while CG, IS, FT and MG applications are used for the overall performance study. This evaluation relies on the methodology described in Section 2.2. In this evaluation, we have some cases of utiliazing only one SPE in Cell processor. Those cases are explicitly indicated in the presented results.

Two different hybrid cache configurations are used. One is HYBRID-speculative configuration as described in the previous chapter. The other one is HYBRID-speculative-pf which is the HYBRID-speculative configuration optimized with automatic prefetching and modulo scheduling transformation. The HYBRID-speculative-pf configuration is the most optimized version of our hybrid software cache in this thesis, using all of the code transformations proposed in Section 5.3.



(a) STREAM kernels 1 SPE



(b) STREAM kernels 8 SPEs

Figure 5.7: Cache overhead distribution for the STREAM kernels.

### 5.4.1 Prefetch effect evaluation

This section measures and checks the effectiveness of the proposed prefetch technique. Prefetch has been applied to the computational kernels in the STREAM application. Figure 5.7 shows the execution of every kernel with 1 SPE (Figure 5.7(a)) and 8 SPEs (Figure 5.7(b)). Execution time is decomposed into actual computation, cache overhead and non overlapped communication. Prefetch affects this last component. The legends in the X-axis identify the versions with and without prefetch. The first thing to notice is that the computation remains constant for both versions in all kernels. For the case of running on one SPE, prefetch succeeds in avoiding stalls associated to DMA transfers, but it also increases the cache overhead component because of the extra executed instructions. This is also happening when running on eight SPEs. The impact of prefetch is different between the two configurations (one and eight SPEs) because the average data transfer time does

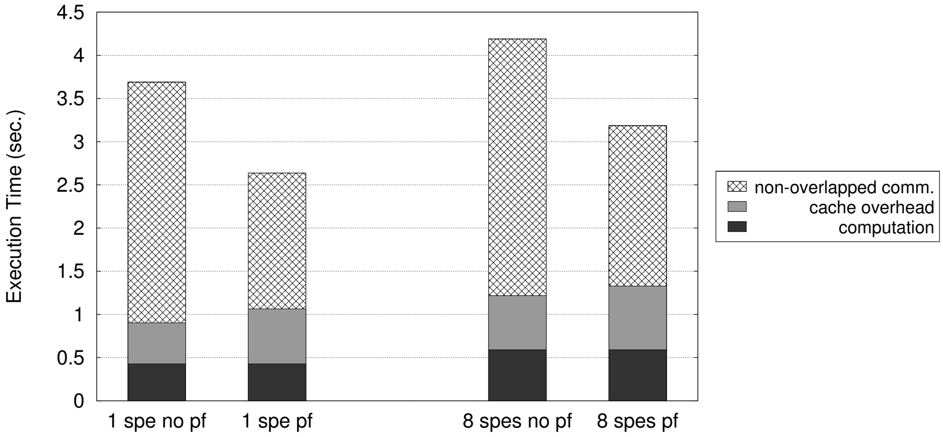


Figure 5.8: Cache overhead distribution for the RandomAccess kernel.

not remain constant. With one SPE, all bus bandwidth is devoted to a single execution flow. In contrast, the bus is shared among all SPEs when DMA requests are concurrently originated in every SPE. This causes the non-overlapped communication to be increased, and the effect of prefetch is more noticeable.

Figure 5.8 exposes the analysis for RandomAccess. Similar results are observed, but now associated to the activity of the Transactional Cache and the modulo scheduling transformations. The execution time is decomposed into the same components that have been analyzed for the STREAM kernels. In this case, both the computation and cache overhead remain at similar levels, as here prefetch is not incurring any increase in the number of executed instructions. The non overlapped communication has a similar weight no matter the number of SPEs, and the effect of prefetch is similar also. This application already saturates the bus with one SPE.

In conclusion, we see that the automatic prefetch technique succeeds in diminishing the stalls caused by DMA data transfers without incurring an unacceptable increase in the number of the executed instructions.

### 5.4.2 Loop Performance

In this section, we evaluate the performance impact of prefetch in our software cache on a per loop basis. The comparison measures improvement in terms of speed-up of the HYBRID-speculative-pf version in respect to HYBRID-speculative version. Figure 5.9 shows speedup factors obtained by enabling prefetch in a variety of loops from CG, IS, FT, and MG applications.

In the case of CG, the improvements range from 1% up to 16% of speedup. The majority of the loops dominated by regular references do not expose significant improvements, usually that is speedup of 3%. Loops 1 and 12 expose a bit bigger speedups of 12% and 14% respectively, since in these two loops the High Locality Cache operates over more references than in the other regular loops of CG and greater synchronization time is addressed by

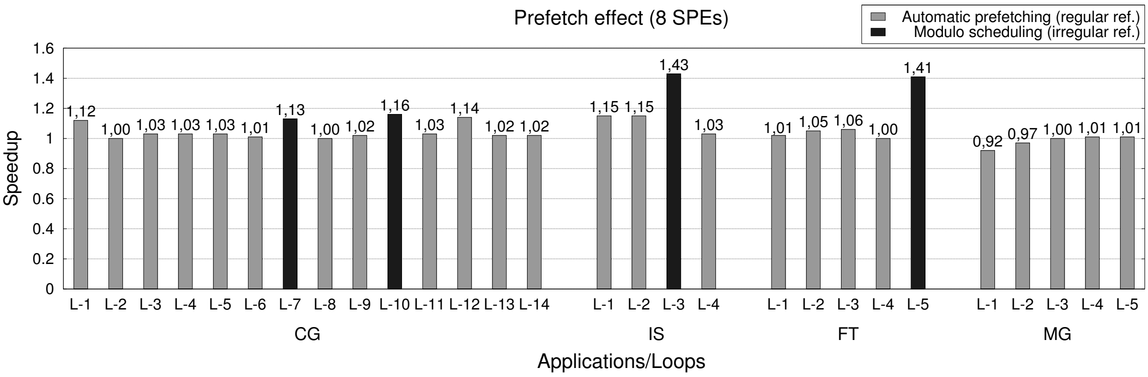


Figure 5.9: Speed-up factors for automatic prefetch and modulo scheduling. Modulo scheduling is used in CG loops 7 and 10, in IS loop 3, and in FT loop 5, due to CG, IS, and FT are totally dominated by irregular memory accesses in the mentioned loops. 8 SPEs are in use.

prefetching optimization. In Section 4.4.3, 15% of synchronization overhead is presented for loop 1 of CG, which is in accordance with the obtained speedup of 12% in this loop. Loops 7 and 10 are dominated by irregular references and the applied modulo scheduling transformation gives an improvement of 13% and 16% respectively. Optimization space for prefetching optimization in CG loop 10 is 22% according to the cache overhead analysis in Section 4.4.3, and modulo scheduling optimized the most of it by giving improvement of 16% in this loop.

In the case of IS application, the benefits obtained in the regular loops follow similar trends as those observed in the CG application (speedups from 3% to 15% for loops 1, 2, and 4), while the benefits obtained from the modulo scheduling in the irregular loop 3 of this application is quite impressive. Introduced modulo scheduling transformation causes a speedup of 43% which is good improvement since, according to the analysis in Section 4.4.3, maximal expected speedup for overlapping communication with computation in this loop is 47%.

The case of FT is different and exposes very poor improvements of up to 6% for regular loops. Only loop 5, affected by irregular references is improved by a factor of 41% caused by the applied modulo scheduling transformation. MG loops are dominated by regular references and they do not improve with applied prefetching or even incur some slowdown, due to increased length of the control code and inefficient prefetching in these loops.

In conclusion, improvements obtained on a per loop basis show that prefetching and modulo scheduling optimizations succesfully targets the non-overlapped communication for the loops in the cache overhead analysis in Section 4.4.3 of the previous chapter.

### 5.4.3 Overall Performance

In this section we present comparison of overall performances. Execution times fo the tested applications are presented in Figure 5.10.

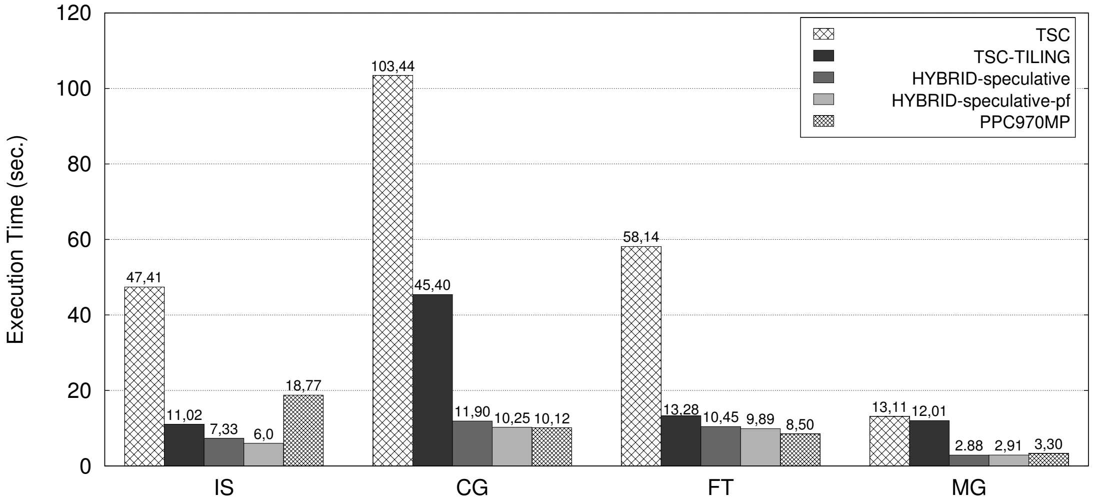


Figure 5.10: Performance of the hybrid software cache design with prefetching optimization and optimized memory consistency support.

Prefetching optimization improves applications dominated by irregular accesses better than applications dominated by regular accesses, due to greater optimization space in irregular applications which is successfully handled by modulo scheduling transformation in our design. So, our best speedups of HYBRID-speculative-pf configuration in respect to HYBRID-speculative configuration are: 1.22 in IS, and 1.16 in CG. FT benefits close to 6% from the prefetching optimization, while in MG speedup factor is 0.99. Improvement of 41% in FT loop 5 (which takes 10% of the total execution time of FT in HYBRID-speculative version), and improvements of 5% and 6% in loops 2 and 3 successfully contribute to the total improvement of 6% in this application. MG is an example of benchmark with a large number of references which occupy almost all storage and does not leave any space for potential prefetching. Besides that, in some MG loops, implemented prefetching is not able to capture actual access pattern and prefetched data is wrong and performance is wasted. Good side is that obtained slowdown in the total execution time of the MG benchmark is just 1%.

Improvement of HYBRID-speculative-pf approach over TSC-TILING is now more noticeable in all loops. Obtained speedups are: 1.84, 4.43, 1.34, and 4.13 for IS, CG, FT and MG applications respectively. We accompany these results with the performance comparison of the HYBRID-speculative-pf and TIILING approaches in STREAM kernels. Obtained bandwidths are shown in Figure 5.11. As expected, the STREAM kernels benefit more from the tiling transformation than from our hybrid software cache since STREAM loops are the best examples where tiling optimization can give the best performance. At least, the gap in performance between TIILING and HYBRID-speculative-pf is not that high as between TIILING and TSC, as described in Section 1.4.

Finally, we observe that the HYBRID-speculative-pf approach is competitive with Pow-

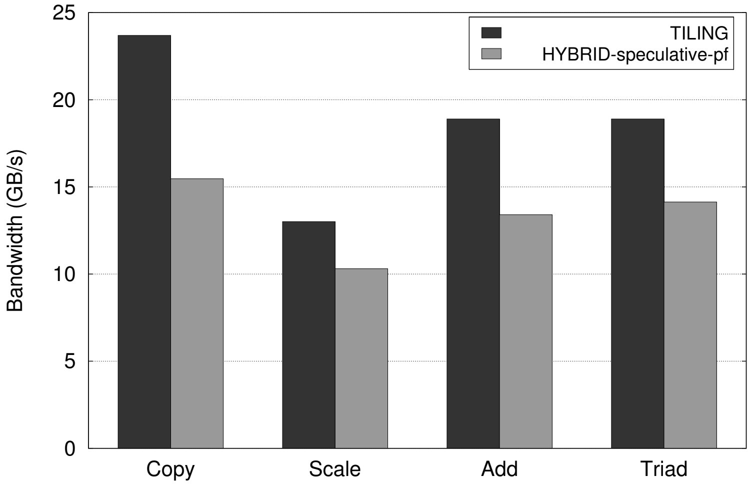


Figure 5.11: Overhead of the HYBRID-speculative-pf approach in respect to tiling with static data buffering approach in STREAM kernels.

erPC 970MP processor in all the tested benchmarks. For IS and CG applications, both Cell BE and PowerPC 970MP suffer from a high degree of communication with main memory. For PowerPC, we know that the working set and access patterns cause a high miss ratio in its cache. The limitation in the local memory capacity in Cell BE has the same effect. In a case of CG, HYBRID-speculative-pf version is close to PPC970MP version instead of being 17% behind when prefetching is not used. Prefetching makes IS to be 3.1 times faster on Cell BE than on PowerPC (instead of previous 2.5 times when prefetching is not activated). Slowdown in MG application causes HYBRID-speculative-pf version to be 1.13 times faster than PPC970MP configuration, instead of previous 1.15 under HYBRID-speculative.

Of all applications, only for FT application HYBRID-speculative-pf performs slower than PPC970MP. Basically, the FT application computes several FFT over a 3-dimensional matrix of complex numbers. The FFT are processed using the following pattern: every plane in a 3-dimensional matrix is evenly cut and every cut is then copied to a temporary variable where the actual FFT computation is done. After that, the output of the FFT is copied back to the 3-dimensional matrix. In the PPC970MP version, the temporary variable fits nicely in cache, thus every FFT computation always hits in cache. In the HYBRID-speculative-pf version, besides all the applied optimizations, FT suffers from a much higher degree of communication because the local memories of the Cell BE are much smaller than the cache of the PowerPC 970MP.

## 5.5 Related Work

The most related prefetching technique is implemented in the IBM's compiler for the Cell BE architecture which uses software caching and tiling transformations with static buffer assignments [37]. Regular memory references are treated by static buffers which are introduced by the compiler, usually involving loop tiling techniques. Prefetching for these memory references is done by double-buffering techniques which are supported by the compiler. However, this approach requires precise information about memory aliasing at compile time which is not the case for our prefetching in the High Locality Cache. On the other side, this approach is well integrated [22] with traditional software cache which is further improved with prefetching techniques for irregular memory references [23]. Prefetching irregular references is based on the inspector/executor model in this approach. Original loop is distributed into inspector and executor loops. Inspector loop collects all addresses to be accessed in the computation. Then, all the collected addresses are used to prefetch data so that needed data is in the local memory once the execution reaches executor loop. In our case we do not split loop execution into inspector and executor loops. Instead, we integrate prefetching and computation by applying modulo scheduling transformation.

Prefetching mechanisms for hardware caches are more widespread [113]. A variety of mechanisms are present there: software controlled mechanisms [20, 63, 66, 77], speculative mechanisms conditioned by a pre-computation in separate execution threads [26, 27], inspector/executor prefetching mechanisms [31, 105], and totally transparent and automatic hardware prefetching mechanisms [55, 82]. However, all the listed prefetching mechanisms rely on the hardware support for data prefetching, which is not the case in our, totally software-based, approach. Similarly to our approach, many of these hardware-based prefetching mechanisms make a disambiguation of different type of access patterns that memory references expose, in order to take advantage of different prefetching polices. The most common memory references that are treated by prefetching mechanisms for hardware caches are: stride references [48, 55, 77, 82], linked references [48, 63, 66, 95, 122] and irregular references[10, 26, 27, 48, 95, 105, 122]. Stride and irregular references are handled in our work, while prefetching of memory references belonging to linked data structures are not focus of this thesis.

Besides general prefetching mechanisms, many domain-specific prefetching techniques are present in research as well. For instance, Memory Hierarchical Layer Assignment (MHLA) [32] is a unified technique which addresses the problem of optimizing the data assignments into memory layers and the block transfers between memory layers. This technique starts from the source code specification of the application and by collecting profiling information optimizes memory mapping and execution order of data transfers. Also, memory organization is potentially customized by this technique. The similarity of this technique with our approach is that prefetching is implemented by invoking DMA operations in order to overlap computation and communication. In contrast to our technique, MHLA is aimed for buffering techniques and simple memory organizations due to application-specific prefetching approach.

Interrupt Triggered Software Prefetching (ITSP) [14] is a prefetching technique for real-time embedded systems that adds prefetching instructions in interrupt handler software to target cache misses. Prefetching optimizations done in ITSP tunes the software to be executed based on observed performance during previous executions. In contrast with our work, ITSP relies on profiling information collected during previous executions of application and hardware prefetching instructions are used.

## 5.6 Conclusions

This chapter describes and evaluates two prefetch techniques for our hybrid software cache architecture. Automatic prefetch is introduced for memory references exposing a high degree of spatial locality, while modulo scheduling is introduced for the memory references exposing no locality and irregular access patterns. The design has been evaluated with a several parallel numerical applications from the NAS benchmark suite and with the synthetic STREAM benchmark. Prefetch was introduced in several parallel loops and speedups ranging from 1% up to 40% have been observed. In two MG loops, some slowdown is observed due to the fact that in the software caching systems prefetching does not come for free, control overhead of executing prefetching code is imposed. In general, implemented prefetching and modulo scheduling transformations successfully improves overlapping of the communication with computation in most of the tested applications. In return, HYBRID-speculative-pf configuration performs better then TSC-TILING configuration in all tested applications. Also, HYBRID-speculative-pf configuration is competitive to PPC970MP configuration in all tested applications.

Finally, overhead distribution from the Section 4.4.3 of the previous chapter and the speedups of the prefeching from this chapter show that the overhead of synchronizing with pending DMA transfers is not an issue anymore. Therefore, the only left issue in the software caching is the essential overhead of executing control code for managing High Locality Cache and Transactional Cache. This control code overhead is a necessity in the systems where buffer management is done through software-based techniques. However, we address minimizing the control code overheads for buffer management of on-chip local memories by some hardware optimizations in Chapter 7.

# Chapter 6

# Realignments On The Fly

This chapter is about our first hardware optimization. Here, we propose a novel DMA engine (DMA++) that breaks alignment constraints imposed on the DMA transfer level by doing data realignments on the fly. This chapter is organized as follows. Summary of the main challenges for dealing with alignment constrains and the motivation for our proposal is presented in Section 6.1. Section 6.2 presents detailed design of the DMA++ engine for realignments on the fly. Section 6.3 evaluates our approach using some real applications. Related work is presented in Section 6.4 and the conclusions are presented in Section 6.5.

## 6.1 Motivation

Multimedia extensions have been successfully introduced in the design of general purpose multi-core processors for some time. Existing multimedia extensions are supported by Single-Instruction Multiple-Data (SIMD) units and they can be found in a variety of processors such as the Intel (IA-32), IBM (PowerPC, SPE), Alpha and MIPS architectures as well as in graphics engines, accelerators in high-performance computing, and Digital Signal Processors (DSP).

SIMD units and their efficiency are usually bounded by alignment constraints related to the memory architecture. Load and store instructions can access only a *vector* of data at the register-length aligned memory addresses (*aligned accesses*). SIMD units are optimized to be efficient for aligned memory accesses while they suffer from a performance penalty for the unaligned memory accesses [28, 103]. SIMD code with aligned memory accesses (*aligned code*) is better in many aspects (performance, power efficiency, and code size) than code with unaligned accesses (*unaligned code*). It has been shown that aligned code is between 1.5 and 3 times faster than unaligned code in a variety of multimedia applications [6, 99].

Two major approaches for handling unaligned memory accesses exist in the current processor designs:

- A variety of hardware mechanisms to perform transparent unaligned load and store operations.
- A special set of instructions used to do the realignment in software.

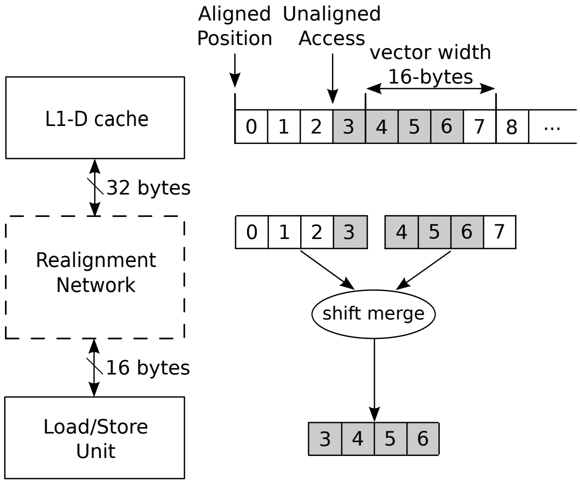


Figure 6.1: Hardware Solution for unaligned memory accesses.

Besides these two approaches, some architectures produce system exceptions that generate calls to the operating system to solve the problem of unaligned accesses. However, some architectures do not have any support for unaligned memory accesses (e.g.: many traditional DSP architectures).

### 6.1.1 Hardware Based Solutions

Hardware based solutions [16, 104, 109, 112] mainly consist of a network that realignes data (a realignment network) in between the load/store execution units and the L1 data cache. In Figure 6.1 we present a hardware based solution where we assumed the vector size of 16 bytes. In order to produce one vector of data, the realignment network, in Figure 6.1, reads two vectors of data executing two aligned memory accesses. The two aligned accesses produce two contiguous vectors of data and after loading these two vectors, the realignment network applies a shift and merge operation in order to produce the requested vector that contains correct data for the original memory access.

### 6.1.2 Software Based Solutions

Software based solutions are used in architectures with no hardware support for unaligned memory accesses. Data for unaligned accesses is generated through a very similar process as the one implemented in hardware based solutions. A set of instructions is executed, comprising different steps: two load instructions generate the two data vectors and then shift and merge instructions are executed to produce the data required by the unaligned access. Merging instructions rely on a realignment token that specifies how to merge the low and high parts of the two vectors [79]. The realignment token consists of a bit mask that either corresponds to the unaligned address (like in MIPS-3D and Alpha) or is produced by special instructions like in Altivec with its Load Vector for Shift Left instruction. In general, software solutions always incur some performance penalty, although the actual

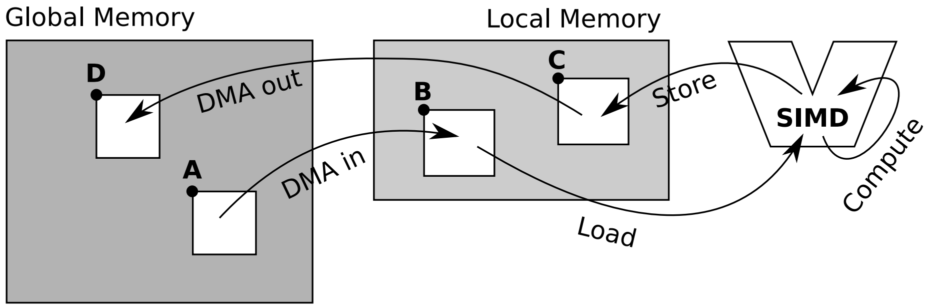


Figure 6.2: Accelerator-based architectures with on-chip local memories.

overhead of the realignment process depends on architectural aspects like the number of execution units in the architecture and the ability of the compiler to appropriately schedule the shift and merge instructions to hide their overhead [38, 119].

### 6.1.3 On-chip Local Memories, DMA Engines, and SIMD Units

In the context of accelerator-based architectures, such as the Cell BE SPE [39, 45, 46], there is one particular aspect of their architecture which enlarges the set of optimizations for SIMD parallelism. In these architectures, every accelerator core combines the usage of SIMD units with on-chip local memories, DMA engines, and a very fast access to the memory subsystem through a high bandwidth interconnection network. These architectures are affected by heavy alignment restrictions, to meet design simplicity and high levels of power efficiency. These restrictions apply to the on-chip local memory access (load/store instructions executed in the SIMD execution unit), as well as to the DMA programming where specific restrictions are imposed in bus protocols and observed in terms of the alignment of the local/global addresses and the amount of transferred data. Current bus protocols (e.g., AXI [7], OCP [11], and a bus protocol for Cell BE [59]) for high-performance architectures define that both addresses used in the DMA transfer have to be equally aligned and the amount of data to transfer must be a multiple of the memory granularity the architecture supports, that is usually the size of the vector registers in the architecture.

In Figure 6.2 we have an example of a typical work to be done in the accelerator-based architectures. Here we have to do a computation on a big matrix in global memory. In order to do that in the accelerator, we have to transfer that matrix in smaller pieces in the local memory. So, in Figure 6.2, first we transfer (DMA-in) a part of the main matrix, pointed to by the starting address $A$, from global memory to local memory at the place pointed to by address $B$. Then we have iteratively to load data in the SIMD unit, to compute, and to store the computed data in local memory at the place reserved for the output matrix (a matrix with starting address $C$ in Figure 6.2). When computation is done, output matrix is transferred (DMA-out) to the global memory at the address pointed to by $D$ in our example. In the explained scenario, two alignment restrictions arises: (1) Addresses $A$ and $B$ in Figure 6.2 must be equally aligned due to DMA engine alignment restrictions. The same restriction applies for C and D; (2) Addresses $B$ and $C$ must be aligned on a vector

```
for(i=0; i<n; i++)
   for(j=0; j<n; j++)
      c[i][j]=b[i][j] + ...
//Here we assume:
//data size of 4 bytes
//vector size of 16 bytes
```

(a) Computation example.

```
for(i=0; i<n; i++)
   for(j=0; j<n; j+=4)
      Rb<-load(&b[i][j])
      ...
      Rc<-add(Rb, ...)
      store(Rc)->&c[i][j]
```

(b) Aligned code.

```


for(i=0; i<n; i++)
   for(j=0; j<n; j+=4)
      Rb1<-load(&b[i][j])
      Rb2<-load(&b[i][j+4])
      Rb<-sh_merge(Rb1, Rb2)
      ...
      Rc<-add(Rb, ...)
      store(Rc)->&c[i][j]
```

(c) Unaligned code.

```
for(i=0; i<n; i++)
   Rb1<-load(&b[i][j])
   ...
   for(j=0; j<n; j+=4)
      Rb2<-load(&b[i][j+4])
      Rb<-sh_merge(Rb1, Rb2)
      Rb1<-Rb2
      ...
      Rc<-add(Rb, ...)
      store(Rc)->&c[i][j]
```

(d) Optimized unaligned code.

Figure 6.3: Source code example.

boundary due to SIMD unit alignment restriction.

Note, the efficiency of the SIMD unit depends on the alignment of the matrix $A$ in global memory, since that alignment has to be preserved during the DMA-in transfer. If $A$ is not aligned on a vector boundary in main memory then $B$ cannot be aligned on a vector boundary also, and we will have to handle unaligned accesses in SIMD unit. For these architectures, the problem of unaligned accesses is solved by pure software-based techniques. Hardware-based solutions are discarded to keep a simple and efficient design. In Figure 6.3(c) and Figure 6.3(b) we can see examples of unaligned and aligned code for the computation in Figure 6.3(a). In a case of unaligned code (Figure 6.3(c)), the programmer or compiler introduces additional load operations and generates appropriate shift and merge instructions assuming particular alignments for load and store instructions. In Figure 6.3(d), we have an example showing how unaligned code can be more optimized for stride-one references [38, 119] by reusing registers in a second shift and merge actions. There, one load is moved in the prologue and one load is in the loop body. A value that is loaded in the loop body is reused in two iterations of the inner loop. In the case of aligned code (Figure 6.3(b)), we have one vector load per reference and the rest of the code is free of the shift and merge overhead. Impact of the unaligned code on a computation has been already studied [6, 99] and we are not addressing this issue in this work.

In the example (Figure 6.3(c)) we assumed that the store address is aligned because we can allocate matrix $C$ at the most appropriate place. But even if we allocate matrix $C$ on the vector boundary we will face the alignment restriction imposed by DMA if alignment $C$ does not match the alignment of the destination address $D$ in the global memory. Handling of an unaligned store operation has a bigger overhead than handling of an unaligned load operation [6].

In order to avoid those alignment problems, it is common to use special allocation functions (e.g. *posix_memalign*) to control the alignments of the data in global memory. In Figure 6.2, when possible, programmers can try to explicitly allocate memory for matrices $A$ and $D$ on the vector boundary. This optimization scheme works fine if it is known at compile time which data is going to be transferted between global and local memory, so we can optimize memory allocation for that data. But if the starting addresses of the data in global memory (addresses $A$ and $D$ in Figure 6.2) are dynamically determined at run time (e.g. in many video codec applications) then using the specific allocation functions cannot help.

Yet, in order to help avoid those problems of dealing with unaligned accesses, there is one unexplored solution based on enlarging the capabilities of the DMA engine. One alternative is to relax the restrictions on the DMA engine, and support DMA commands that involve unaligned memory addresses and an unrestricted amount of data. With this support, we will be able, no matter the alignment of the data in global memory (matrices $A$ and $D$ in Figure 6.2), to choose the most appropriate alignment of the data in local memory (matrices $B$ and $C$ in Figure 6.2). So, among two separate alignment issues in the context of accelerator-based architectures with SIMD units and local memories we face the alignment restrictions imposed in DMA engines by modifying the way the data is treated in the DMA engine as long as transfers are in flight between the local memory and global memory.

The main achievement here is the design of a DMA engine without any alignment restriction. The basis of the design is the inclusion of a realignment logic that operates during the transfer of data. The new design realigns data at a coarser granularity since data will be transferred, shifted and merged at the bus transfer *block* (the largest unit of data that can be transferred with one bus transaction) level, not at the register level as in the software solutions. Data realignment, in the new design, happens in parallel as data is transferred through the bus, hence, ensuring that it has no impact on the block transfer time. The main novelty of the design is the place and time where realignments happen. It is important to emphasize that the software-based techniques are still necessary since we do not face alignment restrictions imposed in SIMD units and it still may happen that an unaligned access occurs in a program. For instance, it may happen that two or more memory references in a program point to the same buffer in local memory but they require different alignments. In this case it would be good to have an option to deal with unaligned data, since it may be unlikely (in terms of efficient local memory usage) to transfer the same data few times in separate buffers in local memory just in order to achieve different alignments. Note that the approach that we propose is an alternative that programmers can use in order to better organize the data in local memory and to avoid unaligned data for SIMD operations as much as possible. To the best of our knowledge, this is the first proposal for realignments placed at the bus transfer level in the context of multi-cores and shared memories. We evaluated the proposal on the simulated multicore architecture with eight accelerator cores. The design is general enough to be introduced in any architecture with SIMD support, on-chip local memories and a DMA engine. We have observed that

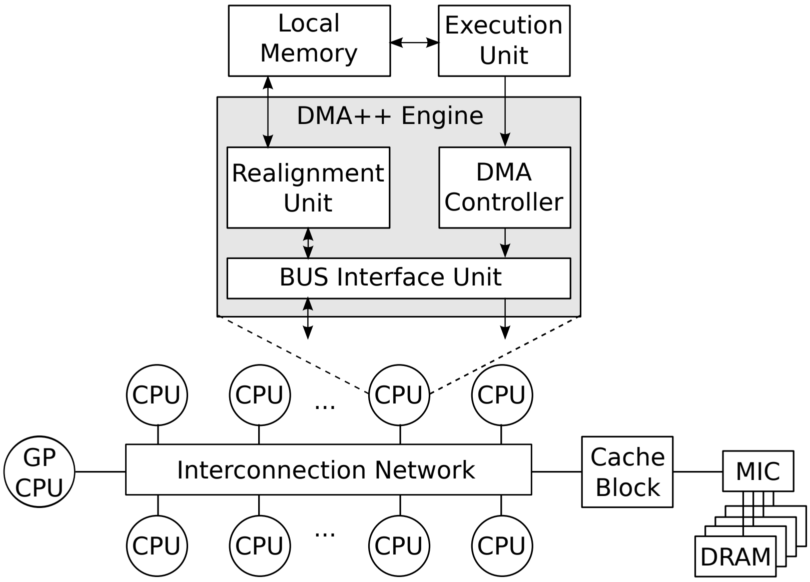


Figure 6.4: Multicore accelerator-based architecture and the DMA++ Engine.

the our DMA design does not incur noticeable overheads in terms of available bandwidth per core.

## 6.2 DMA++ Design

We describe in this section the design of the DMA++ engine for realignments on the fly. Figure 6.4 shows a high level architecture of the DMA++ engine as a part of the accelerator cores in the context of the multicore architecture consisting of multiple accelerator cores (CPU in Figure 6.4) and one general purpose core (GP CPU in Figure 6.4). It is common that the general purpose core (in other words, the master core) is present in this kind of systems in order to drive the accelerator-based cores by a mean of synchronizing and scheduling jobs on them. In the system that we describe in this section we are addressing accelerator-based cores and the DMA++ engine while the general purpose core is mentioned here just in order to clarify who is utilizing the accelerator cores.

In contrast to a traditional DMA engine, the DMA++ engine is enriched with a Realignment Unit. The Realignment Unit introduces new structures and actions in the operational model of the DMA engine that affect how the data blocks are transferred from/to global memory and the local memory. In our proposal, data is realigned in the DMA engine along with the ongoing data transfers (see Figure 6.5) and realignment actions are performed at the block granularity. We can see in Figure 6.5 that the first and the last blocks of the transfer, namely the head and tail blocks respectively are partially occupied with useful data. Atomic merge actions are required for the head and tail blocks in order to maintain memory consistency. The rest of the blocks, namely the body of the transfer, are not affected by this requirement. The realignment process needs all the blocks to be treated in

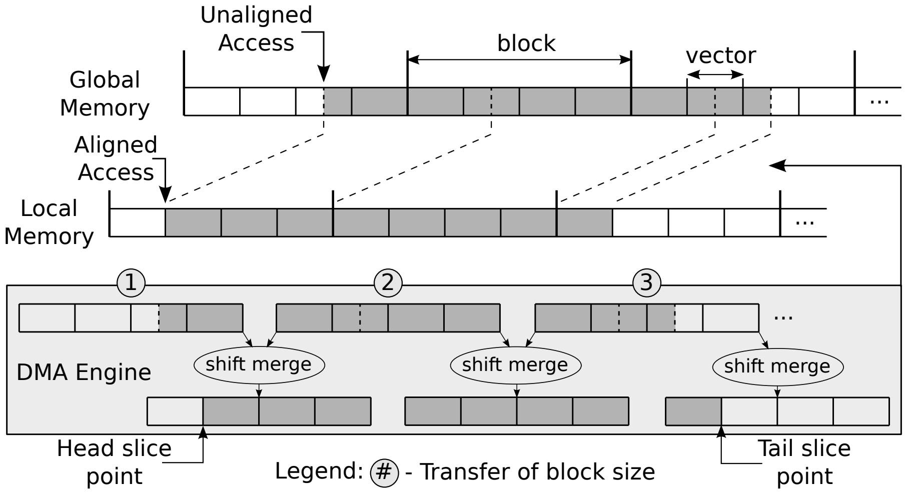


Figure 6.5: DMA transfers with realignments on the fly.

sequential order. Yet, even if we normally expect blocks to come in the requested order, blocks might arrive to the DMA engine under some degree of disorder (e.g.: block *B2* could arrive before block *B1* even if block *B1* was requested before block *B2*) due to latencies in the Interconnection Network, Cache miss penalties, and DRAM conflicts in global memory banks. Accordingly, we define the *disorder degree* as the distance between the expected block (first missing block in the requested order) and the received block. We propose the introduction of a reorder buffer to ensure sequential order at the moment of the realignment. Figure 6.5 shows how data blocks are reused across the shift and merge actions. On the other side, notice that disorder cannot appear when the transfer originates at the local memory.

In order to present the operational model of the proposed DMA engine, we introduce structures and characteristics of the components that appear in the DMA++ engine. The presented set of structures support the most basic functionalities in DMA engines. In the rest of the section we use $V$ and $B$ to denote the vector and block lengths (in bytes) respectively.

### 6.2.1 The Structure Of The System

The DMA++ engine consists of a DMA Controller (DMAC), a Bus Interface Unit (BIU), and a Realignment Unit (RU). The DMAC is connected to the Execution Unit and the BIU. The BIU is connected to the DMAC, the RU, and the Interconnection Network. The RU is connected to the BIU and the Local Memory. Interconnection Network connects all accelerators to a Cache block and Memory Interface Controller (MIC).

#### 6.2.1.1 DMA Controller

The main structure of the DMA Controller is a DMA command table. The DMA command table contains DMA commands, and each entry in the table has 4 fields: (1) the local address, (2) the global address, (3) the size of the transfer, and (4) a DMA command descriptor for the direction of the data transfer. For realignment purposes, we do not support DMA transfers less than $B$ bytes, otherwise this would cause the head and tail to fall into the same block. In the case of a transfer of less than B bytes, an exception is generated. In the evaluation section we show that this is not a significant limitation for most common applications. The maximum transfer size must be a multiple of $B$ and we use $M$*$B$ to denote this size. No other restrictions apply regarding the size of the data transfer. The two basic DMA commands are: the *dma_put* and *dma_get* commands. The *dma_put* command transfers data from local memory to global memory while *dma_get* does it in the opposite direction.

In addition to these two commands, we define two atomic primitives: *get_and_lock* and *test_and_put*. The *get_and_lock* command is conceptually used in conjunction with a subsequent *test_and_put* command to emulate a read-modify-write operation on a specified block in global memory. This feature is already supported in standard DMA engines, for instance the Cell BE processor includes similar support [59]. Our proposal needs atomic commands for merging actions used for the head and tail transfers.

#### 6.2.1.2 Bus Interface Unit

The main structure in the Bus Interface Unit (BIU) is a transaction queue where each entry describes one bus transaction. A transaction is a bus transfer of $B$ or less bytes of data. According to this, BIU supports naturally aligned transfers of 1, 2, 4, ..., $V$, and multiple of $V$ (up to $B$) bytes of data. Each entry in the transaction queue has four fields: (1) the physical local address, (2) the physical global address, (3) the size of the transfer, and (4) a bus operation descriptor indicating the transfer direction. The two basic bus operations are: *put* and *get*. The number of outstanding transactions is limited by the size of the BIU queue. We use $OT$ to denote that number.

#### 6.2.1.3 Interconnection Network

The Interconnection Network (ICN) has its data bus width $W$ and the block size $B$ is an integral multiple of this data bus width. Therefore, every block physically arrives in chunks of $W$ bytes and in the best case chunks will arrive cycle by cycle. In the best case when transactions are pipelined through the ICN, the number of cycles between two consecutive arriving blocks is equal to $B/W$.

#### 6.2.1.4 Cache Block and Memory Interface Controller

A Cache Block that appears in between of the Interconnection Network and Memory Interface Controller (see Figure 6.4) is shared among all cores in the system and operates in the way L3 caches operate in the standard shared memory processors. We have found that the Cache Block has a significant impact on some particular parts of the design and in the evaluation we consider a configuration without the Cache Block as well as the configuration with the Cache Block. When the Cache Block is present in the system then every DMA transfer will go first to the Cache Block and only in a case of a miss a memory request will be sent to MIC by the Cache Block. When the Cache Block is not present then MIC is directly connected to the ICN and every BUS transaction goes directly to MIC. MIC supports naturally aligned memory requests of 1, 2, 4, ..., $V$, and multiple of $V$ (up to $B$) bytes of data. *Puts* of sizes $V$ and multiple of $V$ bytes can be directly written to memory using a masked write operation. *Puts* smaller than $V$ bytes need a read-modify-write operation supported in MIC. Requests of $B$ bytes are always served at the maximum speed. *Gets* are simplified and return always a full block of data. In this case BIU has to select the requested data from the returned block of data.

#### 6.2.1.5 Local Memory

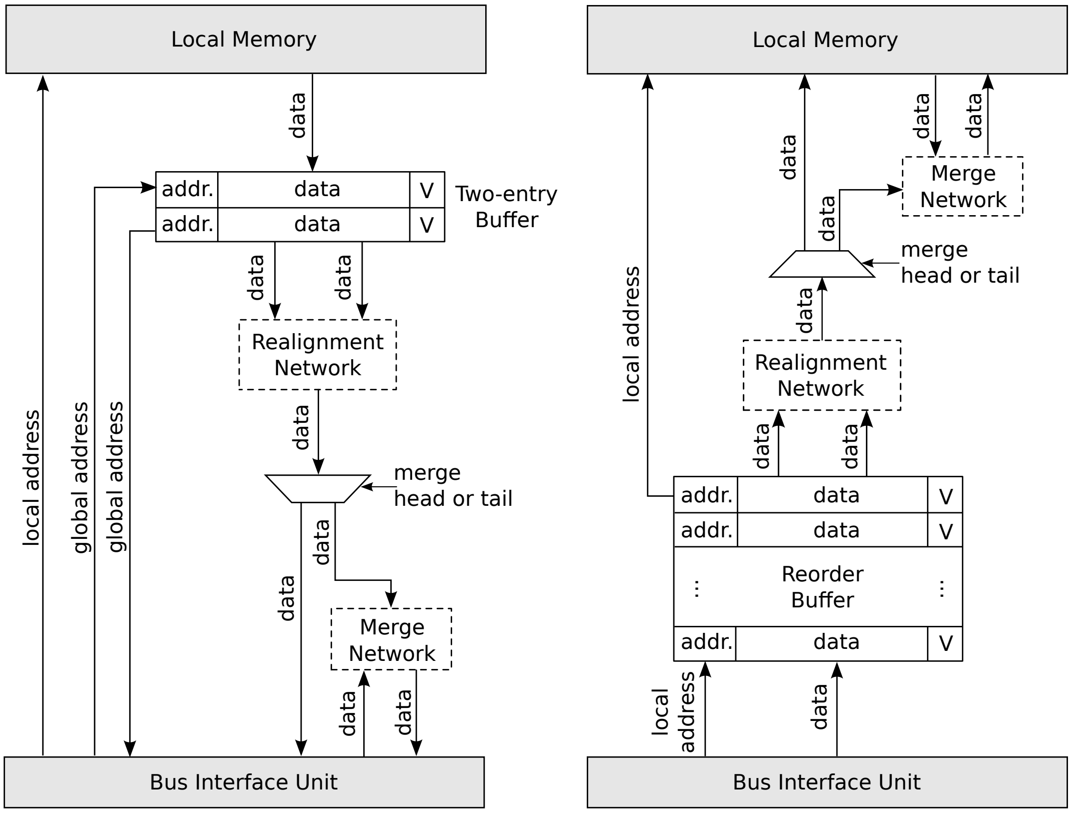
The local memory is accessible by two units: the Execution unit and the DMA++ engine. Because of this, the local memory has at least two ports. The DMA engine can read/write naturally aligned blocks of data ($B$ bytes) from/into local memory in one cycle. Note that in Figure 6.6 we define several paths in the DMA engine to write data to the local memory, but the operational model ensures that only one can be active at a time. Therefore a single port is enough to support access to the local memory.

#### 6.2.1.6 Realignment Unit

The Realignment Unit contains the following structures: a Reorder Buffer (ROB), a Two-entry buffer, Block Decrementers, Realignment Networks, and Merge Networks. The RU design is shown in Figure 6.6 and it has two separate paths, one path for *dma_put* commands (Figure 6.6(a)) and one for *dma_get* commands (Figure 6.6(b)).

The *get* path contains the reorder buffer where incoming blocks are buffered. Every entry in the reorder buffer has three fields: (1) the physical local address, (2) the data, and (3) a valid bit. The reorder buffer supports three basic operations: (1) the writing of the transferred block into the buffer, (2) reading of the two first blocks from the head of the buffer, and (3) shifting of the contents of the buffer one entry towards the head of the buffer. The head is assigned to entry 0 of the buffer. The *put* path is slightly different in terms of structure. The *put* path contains a two-entry buffer. The two-entry buffer has the same fields as the reorder buffer and supports the same basic operations as the reorder buffer.

The Realignment Network consists of a shift count register, and a $B$-byte wide Barrel shifter that shifts data on a byte granularity with the maximum shift of *V-1* bytes. Shift count is determined and stored in the shift count register in the beginning of every transfer. The $\log_2 B$ least significant address bits define the address alignment within the block, namely the offset respect to the nearest address aligned to the block boundary. Shift count is equal to the absolute distance of the destination address offset from the source address offset. If the determined shift count is bigger than *V-1* bytes, an exception is



(a) Put data path.

(b) Get data path.

Figure 6.6: Realignment Unit design (all data lines have width of block size).

generated. Shifting up to more then *V-1* bytes requires more hardware and it would be waste of the resources since the vector width is *V* bytes and shifting of up to *V-1* bytes is enough to achieve proper alignment for the SIMD unit. Note that if there are two separated realignment networks, for the *get* and *put* paths, then RU can handle simultaneous *dma_get* and *dma_put* commands. In the evaluation we are addressing the design where only one DMA command can be active at a time.

In support of the merging process (Section 6.2.2.5), the head and tail slice registers are introduced in the Merge Network. The head slice register contains the address of the first useful byte in the head transfer, while the tail slice register contains the address of the first byte in the tail transfer not involved in the DMA command. The head and tail slice registers identify what parts of the head or tail are modified. In the case of the head, the contents of the block to the right of the head slice point have to be written to memory while contents to the left have to remain unchanged in memory. In the case of the tail, the contents of the block to the left of the tail slice point have to be written in memory while the part to the right remains unchanged in memory. The head and tail slice points are specified in Figure 6.5. The head/tail block base address and offset within the block can be determined by selecting the proper bits from the head/tail slice register.

In addition to the introduced structures, next to the reorder buffer and the two entry buffer there are Block Decrementers that indicate how many blocks have not been transferred. Every time a block is written into the reorder buffer or into the two-entry buffer the associated block decrementer is decreased by one. Every time a new DMA command starts, the appropriate decrementer is initialized to the total number of blocks that are going to be transferred by the command. This total number of blocks is determined during the unrolling process.

### 6.2.2 DMA++ Operational Model

The operational model of the DMA++ engine is organized in five steps: (1) select the command from the DMA command table and perform the address translation, (2) unroll the selected command, (3) issue the transactions from the BIU, (4) realign blocks of the data and write the realigned data to the destination memory, and (5) perform merge actions for the head and tail blocks.

#### 6.2.2.1 Step 1: Command Selection

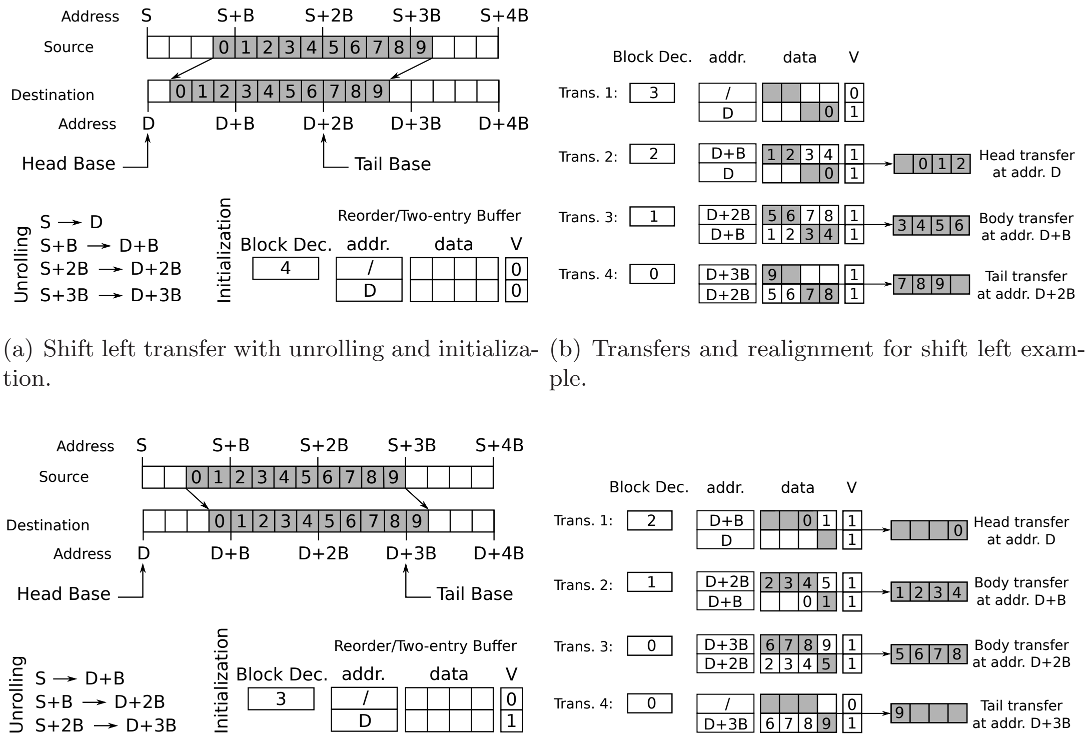
In the first step, the DMAC selects a DMA command from the DMA command table ensuring that only one command can be active at a time. Prior to unrolling, the DMAC has to perform the address translation for the logical addresses in the command. We assume a conventional address translation involving the Memory Management Unit (MMU). When the address translation is completed, the DMA engine enters in step two.

#### 6.2.2.2 Step 2: Unrolling

In the second step, the DMAC starts by computing the shift direction. If the offset within the block (the $\log_2 B$ least significant address bits) in the source address is greater than the offset within the block in the destination address then it has to shift data to the left. If the offset in the source address is less than the offset in the destination address then data has to be shifted to the right. Otherwise, data is equally aligned in global and local memory and the data can be transferred without any realignments.

Details about the unrolling and initialization process are presented in Table 6.1. Here, we define $k$ to be the number of the blocks occupied by the source data: 2 blocks for the head and tail (potentially partially occupied) and $k$-$2$ blocks for body (fully occupied). In the unrolling process, the source and destination addresses are truncated to the nearest block boundary by ignoring $\log_2 B$ least significant bits from the addresses. We use $S$ and $D$ to denote the truncated source and destination addresses respectively. The rest of the unrolling and initialization process depends on the shift direction.

An example of the shift left and shift right cases is shown in Figure 6.7. In Figure 6.7(a) and Figure 6.7(c), we can see how unrolling and initialization is done for our transfer examples according to the rules described in Table 6.1. If it has to merge two blocks of source data starting at addresses $S$ and $S$+$B$ in order to produce a block of the destination



Address S S+B S+2B S+3B S+4B
Source
0 1 2 3 4 5 6 7 8 9
Destination
Address D D+B D+2B D+3B D+4B
Head Base
Tail Base
Unrolling
S → D
S+B → D+B
S+2B → D+2B
S+3B → D+3B
Initialization
Reorder/Two-entry Buffer
Block Dec. addr. data V
4
/
D
0
0
Block Dec. addr. data V
Trans. 1: 3 / D 0 0 1
Trans. 2: 2 D+B D 1 2 3 4 0 1 1
0 1 2
Head transfer at addr. D
Trans. 3: 1 D+2B D+B 5 6 7 8 1 2 3 4 1 1
3 4 5 6
Body transfer at addr. D+B
Trans. 4: 0 D+3B D+2B 9 5 6 7 8 1 1
7 8 9
Tail transfer at addr. D+2B
Address S S+B S+2B S+3B S+4B
Source
0 1 2 3 4 5 6 7 8 9
Destination
Address D D+B D+2B D+3B D+4B
Head Base
Tail Base
Unrolling
S → D+B
S+B → D+2B
S+2B → D+3B
Initialization
Reorder/Two-entry Buffer
Block Dec. addr. data V
3
/
D
0
1
Block Dec. addr. data V
Trans. 1: 2 D+B D 0 1 1 1
0
Head transfer at addr. D
Trans. 2: 1 D+2B D+B 2 3 4 5 0 1 1 1
1 2 3 4
Body transfer at addr. D+B
Trans. 3: 0 D+3B D+2B 6 7 8 9 2 3 4 5 1 1
5 6 7 8
Body transfer at addr. D+2B
Trans. 4: 0 / D+3B 6 7 8 9 0 1
9
Tail transfer at addr. D+3B

(a) Shift left transfer with unrolling and initialization.

(b) Transfers and realignment for shift left example.

(c) Shift right transfer with unrolling and initialization.

(d) Transfers and realignment for shift right example.

Figure 6.7: Shift left and shift right transfer examples with unrolling, initialization, and evolution of the realignment logic during the transfer time.

Table 6.1: Unrolling and Initialization.

| | Shift Left | | | | Shift Right | | | |
|---|---|---|---|---|---|---|---|---|
| Unrolling | op. | src. addr. | dst. addr. | size | op. | src. addr. | dst. addr. | size |
| 1 | get/put | S | D | B | get/put | S | D+B | B |
| 2 | get/put | S+B | D+B | B | get/put | S+B | D+2*B | B |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| k | get/put | S+(k-1)*B | D+(k-1)*B | B | get/put | S+(k-1)*B | D+k*B | B |
| Init. | Write address D in the address field of the head of the buffer. Invalidate all data in the buffer. Block Decrementer is set to k. | | | | Write address D in the address field of the head of the buffer and set valid bit for that entry. All other entries are invalidated. Block Decrementer is set to k. | | | |

data starting at address $D$, then the unrolling process always places address $D$ next to address $S$. This explains why we have differences in the unrolling for shift left and shift right cases (Table 6.1). The unrolling process has to be aware of these differences in order to drive the Realignment Unit (RU) to work correctly. In a case of shift right transfers in order to produce a block of the destination data at address $D$ we have to merge blocks of source data at addresses $S$-$B$ and $S$. Since block at address $S$-$B$ is out of the useful area then the head of the Reorder/Two-entry buffer is initialized as valid containing trash data (Figure 6.7(c)) and the first pair in the unrolling is $S \rightarrow D + B$ instead of $S \rightarrow D$ as in the shift left case. In the shift left case a valid bit for the head of the Reorder/Two-entry buffer will not be set (Figure 6.7(a)). The address field of the head of the Reorder/Two-entry buffer is always initialized with address $D$. The Block Decrementer is equal to $k$ which matches the number of the bus requests generated by the unrolling process. Note that unrolling in Table 6.1 ends up generating only block size requests, which are the most efficiently served in the MIC. Finally, the shift count will be used by the RU to apply the appropriate data realignment.

#### 6.2.2.3 Step 3: Transaction Issuing

In the third step, BIU issues the transactions. Here we have to consider cases for get and put data paths.

In the case of *dma_get* commands (Figure 6.6(b), packets are requested from global memory and after an arbitrary number of cycles, data starts coming to the DMA engine. The transferred data has to be assigned to the specific entries in the reorder buffer. To do so, the RU relies on the physical address in the local memory to select one particular entry. We define the distance between received block and the head of the buffer as the difference between the physical address stored in the head of the buffer and the target physical address. The value of the distance is used to identify the entry to be used to store the arrived data. It might happen that the distance value is outside the range of valid entries. Then, data has arrived under unexpected disorder, and the reorder buffer cannot hold the data as it is waiting for the blocks previously requested. In this case, data has to be discarded and the associated bus request has to be reissued. We found that the Cache Block is an important element with a significant impact on the transaction reissuing and accordingly we use two different strategies to reissue transactions:

- **reissue (without Cache Block)**: In this case we reissue transactions as soon as possible, hence a reissued request has to take the whole path from the DMA engine to MIC and to wait in the MIC's queues along with the other outstanding transactions, until data will be returned to the DMA engine.
- **reissue (with Cache Block)**: Since in this case the Cache Block is acting like a big buffer, then every reissued transaction is most probably going to hit in the Cache Block and data will be provided very fast. According to this behavior, we postpone a reissuing of a transaction as long as there is no available space in the ROB for that transaction. Whenever a place in the ROB becomes available for storing a new transaction, we check whether the appropriate transaction has already been scheduled in BIU queue or not. In the latter case, the transaction is scheduled in the front position of the BIU queue, so as to be the next one to be issued on the bus.

The impact on performance of this behavior is conditioned by the relation of two factors: the number of outstanding transfers at the bus level and the size of the reorder buffer.

In the case of *dma_put* commands (Figure 6.6(a), the BIU provides the RU with the physical local address and the physical global address (see Figure 6.6(a)). The physical local address is immediately used to read data from local memory while the RU uses the global address to determine in which entry from the Two-entry buffer data from local memory is going to be written. The buffer entry is calculated in the same manner as in the case of the reorder buffer. According to the received physical global address and the address that is placed in the head of the Two-entry buffer, the index (0 or 1) of the entry is calculated. The global address is written into the selected entry together with data from local memory.

Note that reorder/two-entry buffers can serve only one active unaligned DMA transfer at a time. While they are occupied by one DMA command they can't accept packets from any other DMA command. This means that we can have only one active unaligned DMA command at time. Along with an unaligned DMA transfer it is possible to have one or more aligned DMA transfers in parallel since aligned transfers do not require use of the realignment unit and there is no resource contention in this case. If we want to support multiple unaligned transfers at a time then one solution could be to replicate the reorder/two-entry buffer as many times as needed. In this case, when we receive or we have to send a package, we will have to determine which buffer holds appropriate data and then to use that buffer. In the evaluation of the proposal we are focused on our initial design in which only one DMA command can be active at a time.

#### 6.2.2.4 Step 4: Realignments

In step four, when the reorder/two-entry buffer contains necessary data, the Realignment Network uses the two first entries of the buffer and according to the shift count, produces a block of the realigned data (see Figure 6.6). The produced block is written in to local/global memory always at the address that is stored at the entry 0 (head) of the reorder/two-entry buffer. After this, the appropriate buffer is shifted. We propose hardware (see Figure 6.8) to ensure that the reorder/two-entry buffer contains the necessary data to be processed by the Realignment Network. If one of the signals (*body*, *merge head*, and *merge tail*) is active then the Realignment Network can consume the data. Otherwise, the data is not ready to be realigned. Note that only one of those signals can be active at the same time, because we support transfers of at least $B$ bytes, which means that head and tail of the transfer cannot fall into the same block. In Figure 6.8, we can see that the Realignment Network works with the head of the transfer if the address from the entry 0 corresponds to the head base address and both entries are valid. The tail of the transfer is considered if the address from entry 0 corresponds to the tail base address and the Block Decrementer is equal to zero. The body of the transfer is considered if neither the *merge head* nor *merge tail* signals are active and the first two entries of the buffer are valid. In the case of the body, realigned data is directly written to the destination memory. Realigning the data in the RU is overlapped (pipelined) with transfers of the other outstanding transactions. All transactions are pipelined in the Interconnection Network and after the initial overhead, in the best case, transactions will start coming, one block every $B/W$ cycles. We expect that traversing the RU could be possible in no more that three cycles: one cycle to write data in the buffers, another cycle to traverse the realignment network and one more cycle to write

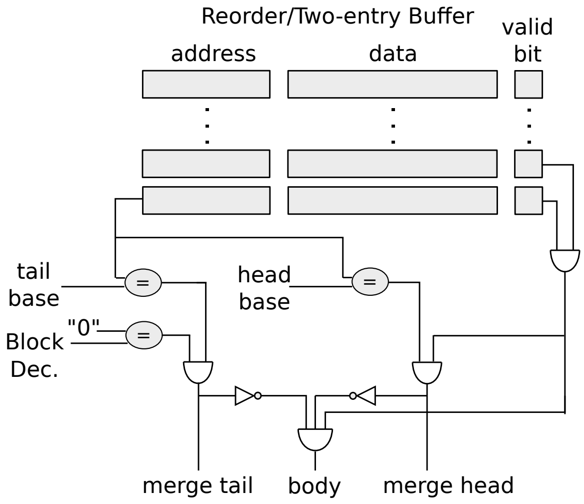


Figure 6.8: Reorder/Two-entry Buffer Design.

data to the local memory or deliver data to the BIU. Besides, during the last cycle buffers are shifted towards the head.

It is important to emphasize that if the number of cycles needed to traverse the RU can fit in the *B/W* cycle window (which is possible, for instance, in the Cell architecture [4]) then traversing the RU will be fully pipelined with the ongoing transfers. In this case the traversing the RU will have almost no impact on the data transfers. In terms of added latency, a delay of 3 cycles (the number of cycles needed to traverse the RU) will be present in the beginning of every unaligned DMA transfer. Overall, an overhead of just a couple of cycles will be introduced on top of thousands of cycles needed to transfer the data involved in a DMA command.

Examples of the realignment logic evolution during the transfer time of shift left and shift right transfer are shown in Figure 6.7(b) and Figure 6.7(d). In these figures we can see how data blocks are transferred and which fields of the Reorder/Two-entry buffer are occupied and how they evolve. Here we can see how unrolling and initialization done in Step 2 drive the RU during the transfer time. Since every realigned block of data is stored to the destination at the address that is present at the head of the Reorder/Two-entry buffer, unrolling and initialization have known this address. The address field of the head of the Reorder/Two-entry buffer is therefore initialized with address *D*.

#### 6.2.2.5 Step 5: Merge for Head and Tail

Step 5 deals with the head and tail cases. We have to distinguish between merge operations for *dma_get* and *dma_put* commands:

- **Merge for** *dma_get* **command**: The Merge Network first reads the appropriate data from local memory, then applies the merge action (according to the context of the head or tail slice register) and finally the merged data is written into local memory.

These three operations read-modify-write must be atomic hence the Execution Unit must not modify the same block in memory at the same time because it could lead to an inconsistency. Note that the atomicity requirement is similar to the one needed for fractional transfers. Therefore, the proposed design relies on already existing mechanisms.

- **Merge for *dma_put* command**: This merge operation repeats three major steps: (1) read the block from global memory using the *get_and_lock* primitive, (2) the merge action (according to the context of the head or tail slice register), and (3) write the block into global memory using the *test_and_put* primitive. If *test_and_put* fails then these steps must be repeated until the *test_and_put* succeeds. This case is not as simple as in the previous one. We require micro-code control to implement the repetition of this process until the update of memory succeeds. Besides, this process might be affected by other execution flows that concurrently are updating the same data block in global memory, potentially increasing the transfer time for the head and tail. To mitigate the effect of this issue, one option is to schedule in parallel the transfer of the body blocks with the transfer of the head and tail.

## 6.3 Evaluation

In the evaluation we consider the system in Figure 6.4 with the following parameters: block size (B) is 128 bytes, data bus width (W) is 16 bytes, maximal DMA transfer size (MB) is 16K, and vector size (V) is 16 bytes. All simulations are performed with the TaskSim simulation infrastructure (Section 2.3.1), which we extend with the DMA++ design, as described in Section 6.2.

Table 6.2 shows the configuration parameters used for the simulation. In general, we have two configurations: one when the Cache Block is used and one without the Cache Block. All elements connected to the Interconnection Network (ICN) have bandwidth of 25.6 GB/s. Since in our applications every DMA communicates only with global memory (through the Cache Block, when present), then our ICN supports two communications in one cycle, one going to memory and one responding from memory to one of the cores. The Cache Block implements a 2MB 4-way set-associative cache. One MIC and DRAM with 4 channels are used. In our simulations, the master processor does not consume or produce data, and it is only used for synchronization.

The evaluation of the new DMA engine design considers the modifications that are made in the DMA engine and studies how data transfers are affected by new functionalities. The next two sections describe applications that are used for the evaluation and describe all aspects of the DMA engine evaluation.

### 6.3.1 Applications

The following applications are used for the evaluation: CG, FT, MG, STREAM, Matmul, SparseLU, Cholesky, Kmean, and Knn. All of them are introduced in Section 2.1.

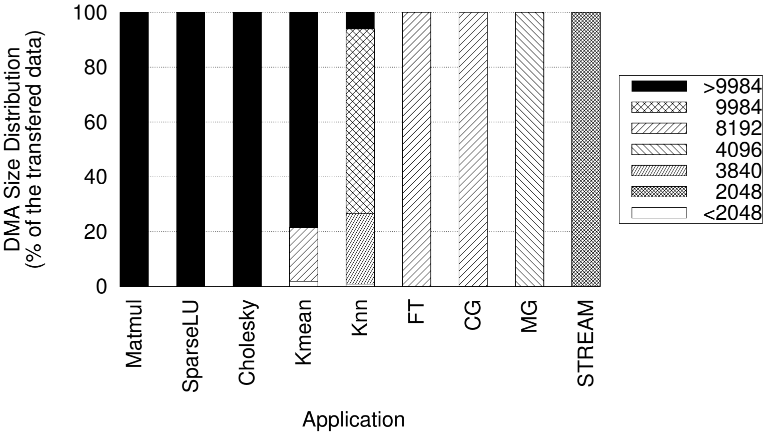


Figure 6.9: DMA size distribution.

For the evaluation of this proposal a bit different set of applications is interesting in contrast to the applications used for the software caching contribution. Here, it is important to cover wider range of DMA transfer sizes in the applications. Thus, all the used applications are hand-coded for the Cell processor to introduce the necessary DMA transfers, and then the applications are executed on the Cell processor in order to get TaskSim traces. Figure 6.9 characterizes used applications in terms of the size of DMA transfers. We can see that presented applications are dominated by a variety of the DMA transfers. Matmul, SparseLU, and Cholesky use only big DMA transfers, greater than 9984 bytes, usually 12K and 16K. FT and CG are dominated by 8K transfers. Kmean and Knn mix both, big and small DMA transfers. MG is dominated by 4K transfers while STREAM benchmark is coded to exploit small DMA transfers that are equal to 2K in this case. We classify these applications according to the DMA transfer size distribution since we noticed

Table 6.2: TaskSim configuration parameters.

| Parameter | Description |
|---|---|
| Cache Block | When enabled it is: 2MB, 4-way set-associative cache |
| DMA engine | 16-entry DMA command table, DMA block size of 128B<br>1 DMA transfer at a time, maximal DMA transfer size of 16KB<br>Number of the Outstanding Transactions (16, 32, 64, 128)<br>ROB size (8, 16, 32, 64) |
| Interconnection Network | Two communications in one cycle, maximum number of concurrent transfers equal to the number of links<br>1 cycle link latency, link bandwidth of 25.6 GB/s<br>Data bus width of 16B |
| Memory Interface Controller | 128-entry queue, data interleaving mask is 4096<br>4 DIMMs |
| DRAM DIMM | 800 million transfers per second<br>autoprecharge enabled, 8 bursts per access |

that in our design the transfer size is important for the disorder and blocks reissuing that may happen in the Realignment Unit. We expect that bigger transfers can result in higher disorder degrees. In the rest of the evaluation, all transfers mentioned here are forced to use the Realignment Unit. We want to test the worst case scenario in which the Realignment Unit and Reorder Buffer are saturated by all transfers in the applications.

### 6.3.2 DMA Engine Evaluation

The most critical issue for the DMA evaluation is the command reissue that appears due to the new DMA design. This depends on the amount of disorder that is observed along the execution of the bus commands. In particular, the parameters that directly affect the number of reissued bus commands are the size of the ROB, the number of outstanding transactions in the bus, namely, the size of BIU queue, and finally the amount of observed disorder.

In order to explore the relation between the three parameters, we have defined the following methodology:

- First, we measure the observed packet disorder in the traditional DMA (no ROB and no packet reissue) on a set of applications using different values for the number of outstanding bus transactions. We expect the outcome of this study to be the maximum disorder degree, giving us an upper bound for setting the potential size of the ROB.

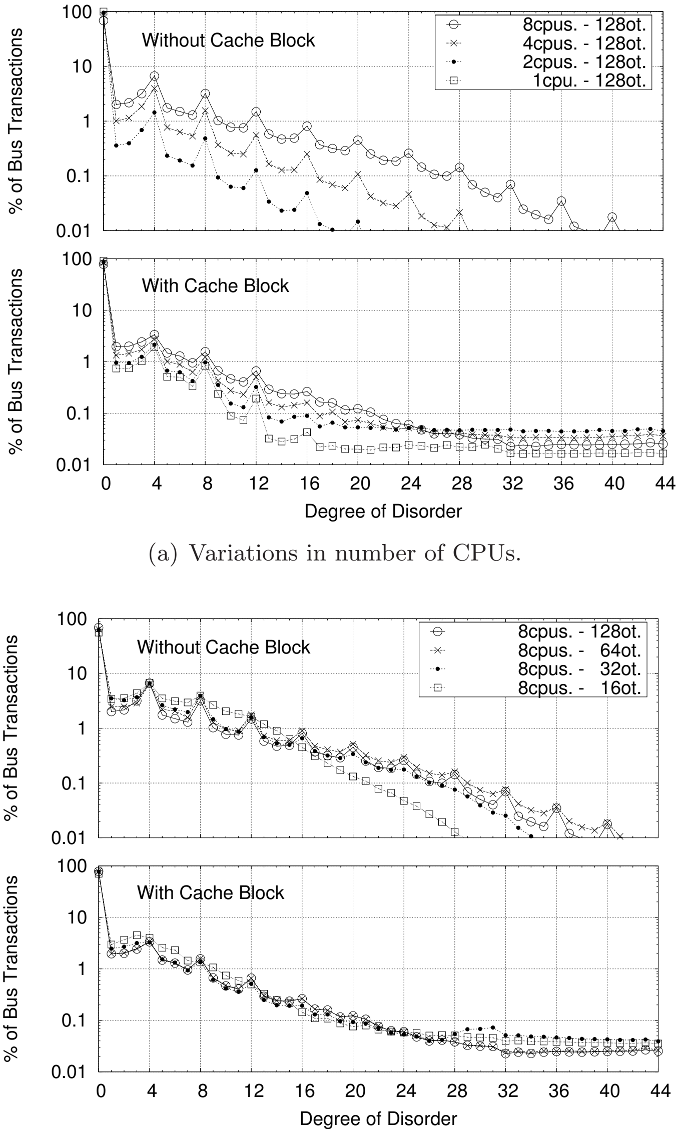
- Second, we account for the number of reissued packets under different configurations of the ROB and BIU. The size of the ROB is set to different values in the range defined in the previous step, and the number of outstanding transactions varies in a pre-defined range. We expect the outcome of this study to determine the optimal configuration that minimizes the number of reissued packets.

- Finally, we study the actual bandwidth per core under the optimal configuration and we show that bandwidth is not affected by the introduction of the realignment logic.

Since the Cache Block has an impact on the observed amount of disorder then all simulations are performed under two configurations, one with the Cache Block and one without the Cache Block. The following three sections describe the evaluation process for the new DMA design.

#### 6.3.2.1 Observed Disorder

The disorder degree in the arrival time of transferred blocks depends on two main factors: the number of outstanding transactions at the BIU level, hereby denoted OT, and the number of cores that concurrently transfer data. We have analyzed the Matmul application under different configurations for these two parameters. During the simulations, we collected data on how many blocks arrived at a particular disorder degree for each simulated configuration.



(a) Variations in number of CPUs.

(b) Variations in number of outstanding transactions.

Figure 6.10: Disorder degree in Matmul application.

Figure 6.10(a) presents graphs for the configuration that keeps constant the number of outstanding transactions (OT=128) and varies the number of executing cores. On the X-axis we show the disorder degree and on the Y-axis we can read the percentage of blocks that arrived at that particular disorder degree. This chart presents the effect of the bus and MIC utilization on the observed disorder. Increasing the number of cores causes a decrease in the number of blocks arrived in-order (disorder degree of zero) and an increase in the number of blocks arrived at the higher disorder degrees. For instance, the percentage of transactions arrived at disorder degree of zero for 1, 2, 4, and 8 executing cores is 100, 95,

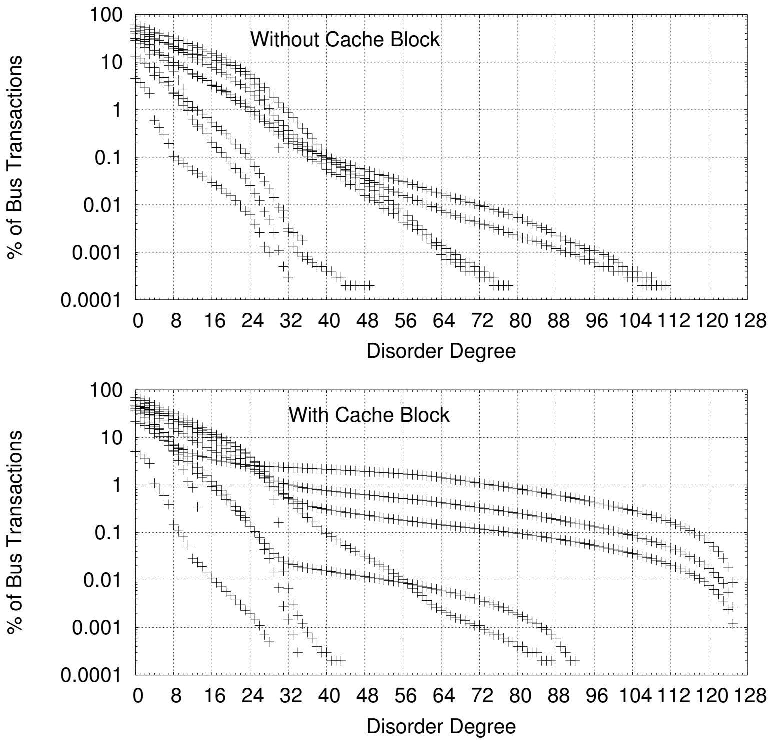


Figure 6.11: Disorder distribution from all tested applications for 8 executing cores and 128 outstanding transactions.

85, and 68 when the cache is not used, and 90, 86, 83, and 78 when the cache is used. For all the other disorder degrees it appears to have more transactions when more cores are executing simultaneously. This distinction is clearly noticeable for all disorder degrees in the configuration without the cache. When the cache is used, the same effect is still noticeable in the tested application but for the disorder degrees less than 21 (Figure 6.10(a)). However, notice that disorder levels greater than 20 account for very few data transfers (less than 0.01% of all transfers) in the tested application.

Figure 6.10(b) presents graphs in order to show the effect of the number of outstanding transactions on the observed disorder, for the maximum pressure in the bus usage when eight cores are executing simultaneously. Now, the number of outstanding transactions does not make a big difference when the cache block is used while some distinction can be seen when the cache is not used. The higher OT causes an increase in the number of transactions at some disorder degrees. For instance, starting from a disorder degree of 18 configuration with 16 OT appears below the other configurations. We observed that the disorder degree is changing faster by increasing the number of cores than by increasing the number of outstanding transactions.

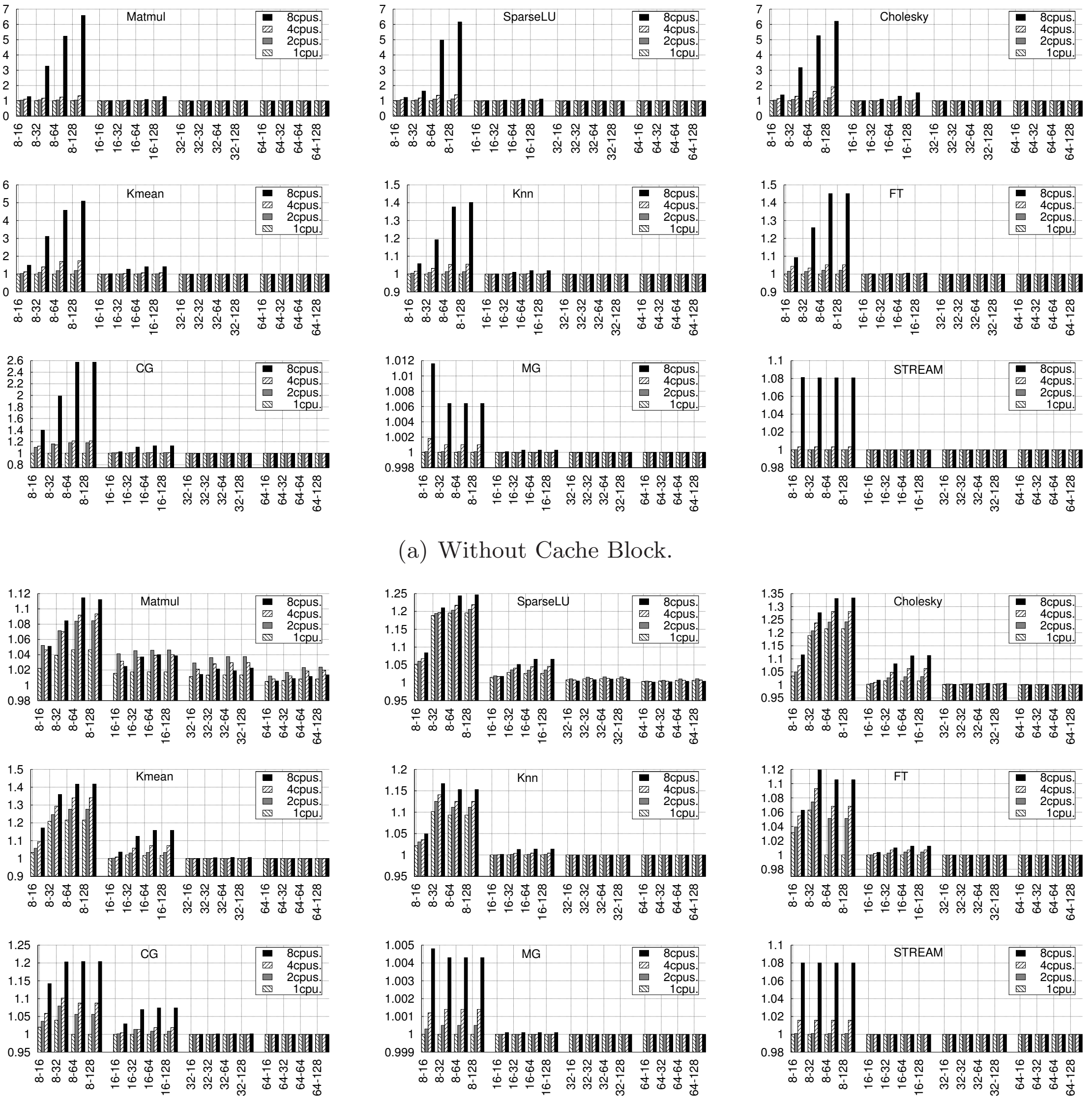
Since the configuration of 128 outstanding transactions and 8 executing cores stresses the system the most, in order to determine the potential size of the ROB, we collected the disorder data from all tested applications for that particular configuration and presented in Figure 6.11. From the charts in Figure 6.11 we can read for any disorder degree *n* on the X-axis, what percentage (Y-axis) of the total bus transactions has arrived at the disorder degrees greater than *n*. Here we do not distinguish between applications, it is only important to see the maximal values for each disorder degree on the X-axis. The ROB of *n* entries can handle all transactions that arrived at the disorder degree less or equal to *n-2*. Therefore, the charts in Figure 6.11 allow us to conclude that ROBs of 8, 16, and 32 entries (look for disorder degrees 6, 14, and 30 in Figure 6.11) will not be able to handle up to 35%, 20%, and 1% of transfers from some of the tested applications when the cache is not used, or up to 40%, 15%, and 2% of transfers when the cache is used. This study does not take into account that reissuing will eventually affect the disorder degree. The next section addresses this fact.

#### 6.3.2.2 Reissuing Overhead

This section measures the reissue effect caused by the relation between the number of outstanding bus requests and the size of the ROB. The main purpose for this is to define the optimal size for the ROB, minimizing the number of reissued transfers and the storage devoted to the ROB. We have studied different configurations for both parameters. The number of outstanding transactions is set to 16, 32, 64 and 128 transfers. The size of the ROB is set to 8, 16, 32 and 64 entries. The charts in Figure 6.12 show the transaction reissuing overhead that has been observed in all tested applications for 1, 2 ,4 and 8 cores. The Y-axis shows the increment factor, where the value 1 stands for having no reissued bus requests.

The most significant overheads are observed when 8 cores are active and the ROB has a small number of entries, just eight. In the worst configuration (8 entries in the ROB and 128 outstanding transactions on the BIU), the reissue degree in the configuration without cache is around 6 (which means that every transactions is reissued at least 6 times) for the applications that are dominated by big transfers (Matmul, SparseLU, Cholesky), while it is from 1.12 to 1.35 when the cache is used. This difference comes from the fact that in the system where the Cache Block is not used we reissue transactions as soon as possible in order to minimize reissuing impact on bandwidth, and it may happen a transaction be reissued more than once. When cache is used, every reissued request is about to hit in the cache and data can be provided fast, thus reissuing action can be postponed as long as there are no available space in the ROB for that transaction. Kmean, Knn and FT experience reissue degrees of 5, 1.4, and 1.45 respectively when the Cache Block is not used while CG, MG and STREAM experience reissue degrees of 2.6, 1.01, and 1.08 respectively. When the Cache Block is used, about 40%, 15%, 20%, 10% of transactions are reissued in Kmean, Knn, CG and FT, while MG and STREAM expose very small reissue overhead in this case (less than 1%).

As expected, a very small reorder buffer can cause a large number of reissued transactions and the bigger DMA transfers we use, the larger the overhead is produced. When we switch to a ROB of 16 entries, the reissue degrees improve significantly and get to a level below 10% in the configuration with cache, and below 50% without cache. For a ROB

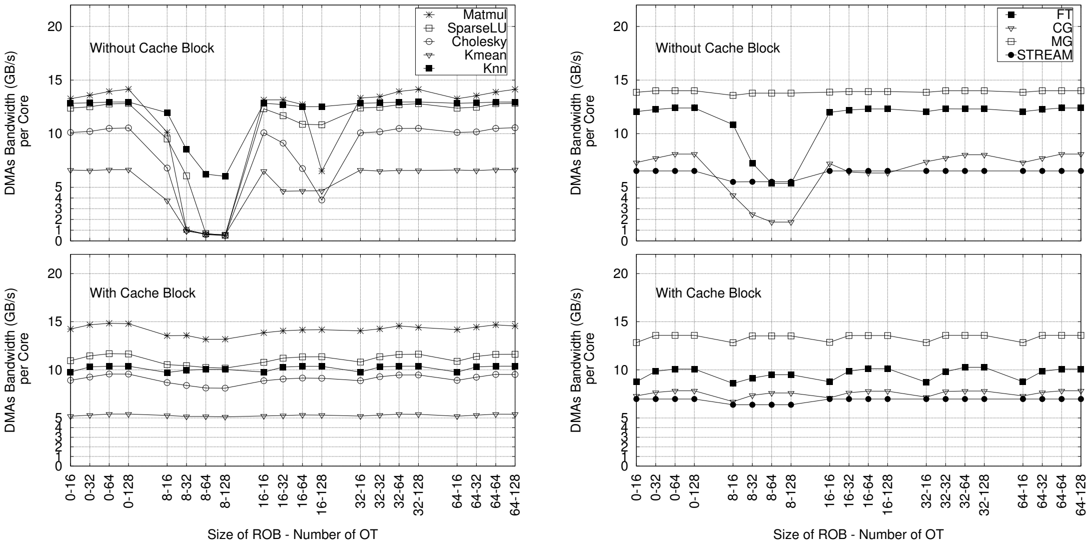


(a) Without Cache Block.

(b) With Cache Block.

Figure 6.12: Transactions reissuing overhead (X-axis: Size of ROB - Number of OT, Y-axis: Relative Number of Issued Transactions).

of 32 or 64 entries, the reissue degree is close to zero, having a negligible effect in both configurations. Therefore, we conclude it is possible to find sufficient size of the ROB, at least if just the reissue degree is taken into account (e.g. sufficient size of the ROB in the tested applications is 32). In general, the behavior that is observed for the 8-core case is observed for the cases where 1, 2 and 4 cores are executing. The observed trends are the same, only the absolute numbers of the reissue degree change. The next section measures



(a) Matmul, SparseLU, Cholesky, Kmean, Knn.

(b) FT, CG, MG, STREAM.

Figure 6.13: Bandwidth.

the impact of the bus requests reissue in terms of actual bandwidth.

#### 6.3.2.3 Bandwidth

In terms of performance it is important to know the impact of the studied parameters on bandwidth. In the previous sections, we have observed that a ROB with 16 entries is not as efficient as a ROB with 32 entries. Yet, it is not clear how both configurations differ on the achieved bandwidth per core. The achieved bandwidth by a core is determined by the total number of bytes transfered by the DMA engine of that core during the execution time and by the number of cycles needed for all those transfers. In this section, we present average bandwidth obtained per core (calculated as the arithmetic mean of the bandwidths achieved in every core) in a system with 8 cores.

Figure 6.13 shows the actual bandwidth obtained for all studied configurations. On the leftmost parts of the four presented charts a baseline configuration (size of ROB is equal to zero) appears. This configuration corresponds to an ideal case where no packet needs a reissue. We denoted that the size of ROB is zero in this case and in the simulator we configured to have an ideal ROB that can handle any disorder degree so that benchmarks can be executed at the maximum speed with no reissue overhead. In the rest of the configurations we have variations in the size of reorder buffer and in the number of outstanding transactions.

Clearly, when the Cache Block is used, the bandwidth curves do not expose any significant variations as long as the size of the ROB is increased, only a ROB with 8 entries reduced the bandwidth for about 10% at most. When the Cache Block is not used, we discard a ROB with 8 entries, as we see that the bandwidth is drastically reduced (up to factor of 20) while for a ROB with 16 entries bandwidth is reduced for 10%-60%. For more that 16 entries in the ROB, the bandwidth curves do not expose any significant variation. In the tested applications a ROB with 16/32 entries is sufficient in the configuration with and without cache. As a side effect of this experiment, we observed that bandwidth of some applications can be improved when the Cache Block is not used (e.g. SparseLU, Kmean, Knn, FT).

## 6.4 Related Work

Two major trend lines exist for handling misaligned memory accesses. On one side, pure hardware-based solutions, that include hardware mechanisms to perform transparent unaligned load and store operations. On the other side, pure software-based solutions based on a special set of instructions used to realign data.

Hardware-based solutions generally work at the level of the memory access granularity of the architecture, and they all have in common the introduction of a realignment network that is activated in the presence of a misaligned access. The realignment logic is placed between the Load/Store execution units and the L1 data cache. In the presence of a misaligned reference, an extra memory access is triggered to produce the required data for the misaligned access. This process consists of shift and merge operations that combine the data produced by the two accesses. This or similar hardware support is used in the SSE3 extensions providing the LDDQU (Load Unaligned Integer 128 bits) instruction that performs a 32-byte load in order to extract the corresponding 16 bytes of unaligned data [16]. This instruction potentially can result in performance degradation, what has been discussed in [53]. The Trimedia TM3270 processor has support for unaligned load and stores with no performance penalty [112]. The TMS320C4X processor (developed by Texas Instruments) [109] and MIPS64 architecture [104] also support unaligned loads and stores.

Software-based solutions rely on the use of specific instructions in the ISA that support the shift and merge operations already mentioned in the context of pure hardware-based solutions. This is the case of the Altivec extensions in the PowerPC [34] architecture, or the MIPS-3D [106] and Alpha [102] architectures. The merging operation relies on a special instruction to merge low and high parts of two vectors into a third one according to a realignment token [79]. In software solutions, it is not possible to ensure that no performance penalty is going to be introduced. The final result depends on many factors, such as the ability of the compiler or programmer to correctly schedule the instructions and hide the shift and merge operations. Eichenberger et al. [38] describes several compilation techniques for stride-one memory references. Their data reorganization is very efficient when they have compile-time information about misalignments. Runtime alignments in [38] use zero shift policy, similar like in a few prior works [24, 30]. Wu et al. [119] improves runtime alignment handling and exploits other policies, the eager shift and lazy shift policies. Larsen et al. [60] exposes a very detailed discussion of memory alignment considerations. This work presents a technique to detect memory alignments and focuses on maximizing the number of equally aligned references in a loop. It uses a variety of techniques (loop unrolling, loop peeling, padding multidimensional arrays, etc.).

An enhanced DMA controller is proposed by Payá-Vayá [83]. It is specially designed for specific VLIW media processors aimed for video applications. The DMA controller is proposed taking the video transfers characteristics into account. The operational model is designed for special requirements - transferring matrices of pixels. Authors emphasized padding issues in video applications and proposed a hardware approach for padding in DMA controller. It is not explained how the enhanced DMA controller deals with blocks that arrive out of the requested order (we referred to that as disorder). In contrast to the enhanced DMA, the DMA++ is aimed at general purpose multi-cores equipped with DMA engines and local memories (e.g. Cell, Intel Larabbee [97]) tailored for accelerator-based computing of critical application kernels.

## 6.5 Conclusions

In this chapter, we have described the hardware proposal for dealing with unaligned memory transfers in modern high-performance accelerator-based architectures with local memories and DMA engines. We took advantage of the DMA transfers needed in the accelerators with local memories, and designed the Realignment Unit that performs data realignment in the DMA engines while data is in transfer between local and global memory. In this approach, the mentioned realignment happens in parallel with data transfers and imposes a negligible overhead on the block transfer time. With this support, programmers can move data from any alignment in the global memory to the most convenient alignment in the local memory and vice versa. This hardware offers a freedom of arranging the data in the local memory that has an immediate impact on improving the quality of the SIMD code in the accelerators by avoiding unaligned data in local memory due to better memory management offered by new functionalities.

We have evaluated our design, and the observed measurements show that the Realignment Unit can be introduced in the DMA engines with no performance loss in terms of available bandwidth.

# Chapter 7

# Accelerated Buffer Management

In this chapter, we present a novel DMA controller (DMA-circular) that accelerates buffer management for on-chip local memories by mapping some functionalities of our software cache into DMA engine. The aim of this chapter is to disuss overheads of software cache approach and to offer an option to accelerate such overheads. Organization of this chapter is as follows. Motivation for the proposal is presented in Section 7.1. Section 7.2 describes the design of our enhanced DMA controller and its programming interface. Section 7.3 evaluates the design. Section 7.4 discusses some related work, and finally Section 7.5 concludes the chapter.

## 7.1 Motivation

As introduced in the first chapter of this thesis, on-chip local memories impose programming difficulties since they are completely managed by software through programmable DMA engines. The best performance is obtained when local memories are managed manually by tuning every DMA transfer in conjunction with a good knowledge of the applications' behaviour. This option has been very successful in the High Performance Computing (HPC) and the embedded domain but is not viable in scenarios where programmability is important. Therefore, besides obtaining good performance, for the acceptance of local memories it is important to overcome the non-transparent/explicit memory management they require, getting closer to a level of programmability where little or no information about applications' behaviour is necessary to automatically adapt the context of local memories to the applications' needs.

So far, software caching has solved the programmability issues but at the cost of some performance penalties. We have seen that our hybrid software cache, even highly optimized, ends up in significant overheads for both irregular and regular memory references (Section 4.4.3). Accelerating control code related to memory management for irregular memory references is out of the contex of this thesis, since any improvement for this type of references would lead to having typical hardware caches in contrast to local memories which are in the focus of this thesis. Interesting overheads to be addressed are those that happen when managing regular references. Regular references are typical references to be handled in local memories and in terms of HPC, it is desirable to minimize control code overheads for dealing with this type of references. In our hybrid software cache, regular references are highly optimized: software lookups are removed from the inner most loops, memory consistency mechanisms are optimized, and automatic prefetching is supported. Anyhow, some overheads still remain and those overheads come from the mapping of the memory references to the on-chip local memory. As we have seen in the introductory chapter in the section about programmability (Section 1.3.2), the essential action of the mapping is to program DMA transfers to bring or write-back data, but that is not all. Since DMA engines provide just basic operations to transfer data to/from the local memory, the software is responsible for all the buffer management: buffer allocation, buffer assignment to references, buffer mapping to the local memory and address translation. Buffer management is a complex problem because it requires memory aliasing analyses. Since even complex compiler analyses cannot reliably answer on the memory aliasing question, compilers have to emit control code that checks aliasing conflicts to consistently perform the buffer management. This control code turns to be an important drawback due to high overheads it generates. The main reason for the overheads is the poor support in current DMA controllers for managing local memories, and that is what we target in this chapter.

In this chapter, we propose an enhanced DMA controller tailored for reducing overheads of managing local memories. The main achievement is the bringing of the DMA controllers to a high level extension aimed for specific applications' needs at exactly appropriate places where local memories and DMAs make sense but still end up in unacceptable overheads due to the need for the control code to substitute the lack of advanced data management support in the DMA controllers. We show that our enhanced DMA controller is able to provide high performance by keeping the control code under 15% of the execution time. Also, speedups with respect to traditional DMA controllers range from 1.20x to 1.70x.

## 7.2 DMA-circular Design

This section describes the design, programmability and applicability of our extended DMA controller (DMA-circular).

Figure 7.1 describes the design of the DMA-circular which is build to accelerate typical control actions which occure in our High Locality Cache. Essential parts of the DMA-circular design are three blocks: Memory References Block, Directory Block, and Translation Block.

Memory References Block serves as an input where all references to be handled by new DMA are placed in order to be used by the other components which do actual data management. The most important block for all the buffer management is the Directory Block which is a cache like structure with a table describing all buffers resident in the local memory. Buffer allocation is addressed by splitting local memory space into equally sized buffers (with a size being power of two), which actually resembles cache lines. Therefore, Directory Block contains one descriptor for each buffer (line) in the local memory. Those descriptors are used to establish the mapping of the buffers from the main memory to the local memory. Memory aliasing is simply addressed by associative lookup in the directory block (as it is done in the caches). So, Directory Block serves as cache like structure capable of doing associative lookup in order to find out whether some data is already mapped in the local memory or not, while the local memory is storage for all buffers. Directory Block is capable of automatically scheduling DMA transfers in a case of missing data in the local memory or in a case of writing-back modified data. Directory Block does buffer assignment which effect is reflected in the Translation Block that holds all local memory addresses resulting from the buffer assignments. So, every reference is mapped to some of the buffers in the local memory, and since those buffers are usually smaller than the whole workload needed for the reference, then actual work can run only for some number of iterations - as long as buffers in the local memory can feed the computation. Memory References Block addresses this issue and computes the mentioned number of iterations.

Essentially, extended DMA is aimed automatically to handle all references which are placed in the Memory References Block by bringing data buffers to serve them, placing buffers at the appropriate places in the local memory, handling memory aliasing and proper buffer assignment to references. It provides outputs in terms of the local memory addresses resulting from the buffer assignment, the number of iterations that are possible to execute over the buffers present in the local memory, and the set of tags to be used in the synchronization.

In terms of the programming interface, new DMA controller is brought on a higher level. Inputs are descriptions of references to be handled while the outputs are tags for the synchronization, local memory addresses to be used in the computation, and the iteration space for the computation. In contrast to the basic DMA engines (described in Section 1.3.1), new DMA engine provides a higher level programmable hardware interface designed to offload from software many control actions for buffer management.

In the following sections, we describe the extended DMA design in greater detail.

### 7.2.1 DMA-circular Structures

The extended DMA consists of the Memory References Block, the Directory Block, the Translation Block, the Configuration Block, the Prefetch Block, and two registers named reg-tags and reg-iters. All these components extend basic DMA controller which corresponds to only Transfer Block in Figure 7.1.

#### 7.2.1.1 Configuration Block

This block supports configuring of the DMA-circular regarding the total number and size of the buffers as well as the place where those buffers should be allocated in the local memory. Size of a buffer and the number of buffers determine the total size of storage for the buffers, while the third parameter is the address where that storage should start in the local memory. For example, if someone wants to devote just half of the local memory to be used by DMA-circular then it is possible to configure that with mentioned three parameters.

It is important to emphasize that the number of buffers meet a limitation dictated by the Directory Block. Hence every buffer has its descriptor in the Directory Block than there

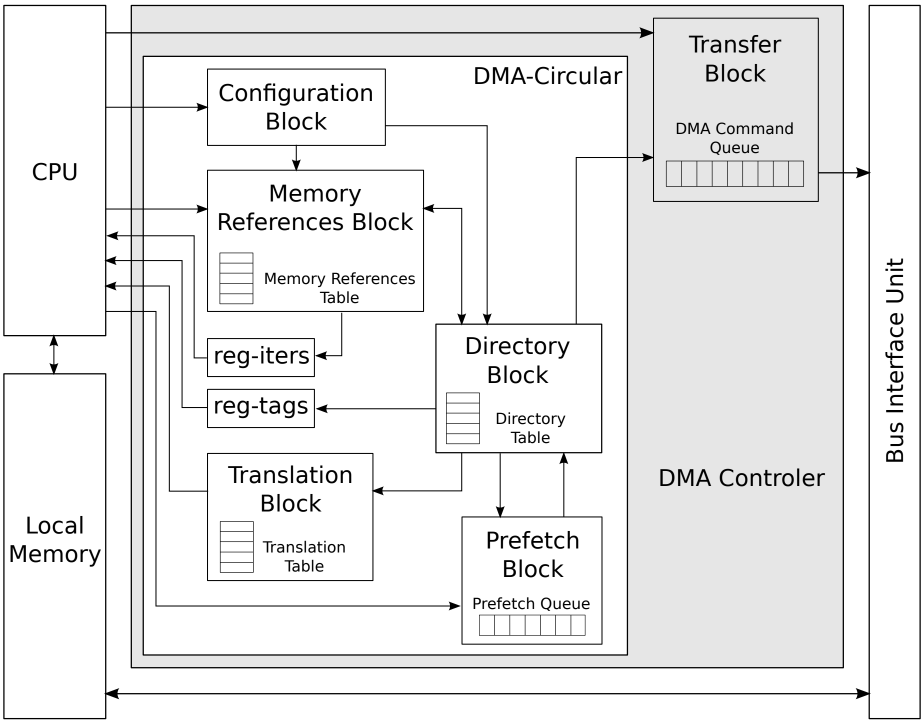


Figure 7.1: DMA-circular high level design.

is a hardware limit for the number of the buffers (similarly to caches and the number of the cache lines). In this work, we use limit of 32 buffers.

#### 7.2.1.2 Memory Reference Block

Memory References Block (MRB) is shown in Figure 7.2. The main structure of the MRB is the Memory References Table (MRT) which contains descriptions for all the memory references to be treated by the new DMA controller. It is important to emphasize that the new DMA controller is tailored to handle references which appear in the loops, but only those exposing *strided access pattern with constant strides* since only for this type of references it is possible to calculate how long (in terms of iterations) a buffer in the local memory can feed the computation. In the MRT, three fields are devoted for description of this type of references: (1) *address* - the address which is going to be accessed by reference in the first iteration of the loop, (2) *stride* - the size of the stride, and (3) *rw* - the read/write information which is used later to figure out whether a buffer assigned to a reference is modified or not. These three fields must be provided by the user while the other two fields (*buffer-id* and *iters*) are maintained by MRB.

Once the hardware is provided with the size of the stride and address of the first element to be accesses by the reference than it is easy to calculate how many iterations it is possible to do within a given buffer of data. The *iters* field corresponds to the iterations while the *buffer-id* field identifies the buffer in the local memory which is assigned to the reference.

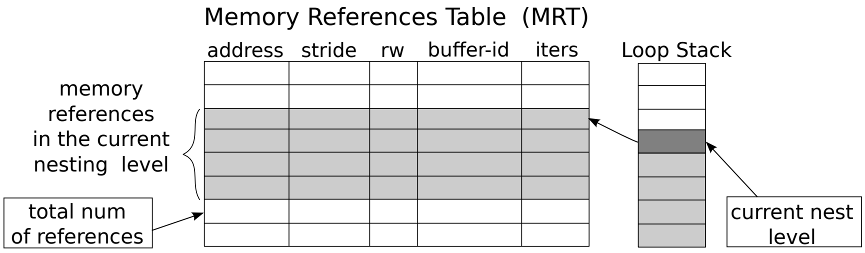


Figure 7.2: Memory References Block.

If *iters* is zero, then we do not have buffer assigned to the reference, or we have reached the end of the currently assigned buffer. Both cases trigger actions in the Directory Block to bring new buffer of the data for the reference.

Loop Stack structure that appears in Figure 7.2 is used for maintaining references in a case of nested loops. We comment more on this in Section 7.2.2.

#### 7.2.1.3 Directory Block

Directory Block (DB) is shown in Figure 7.3. We can see that this block is similar to the organization of standard caches, but extended with some specific features targeting buffer management for local memories. The main structure of the DB is the Directory Table (DT) which contains descriptors for all buffers in the local memory. First entry describes first buffer, second entry second buffer and so on and so forth. Each buffer is described by four fields in the DT (see Figure 7.3): *baddr*, *counter*, *dirty*, and *pf* field. The *baddr* field is the base address of the mapped buffer from the main memory. The *counter* field is a counter that keeps track of the number of references where the buffer is assigned to. Basically, this means that *counter* number of references are using this buffer. The *dirty* field describes whether the buffer is modified or not. The *pf* field specifies if the buffer is under prefetching.

The main purpose of the DB is to provide correct buffer assignment to references, and to handle memory aliasing. This is done in a cache manner, similarly as in our software cache but here it is done in hardware. When some buffer is needed for a reference, then associative lookup is performed in the DT. If a hit occurs, then buffer is simply assigned to the reference by updating its *buffer-id* field in MRT. Also, the *counter* field of the buffer is incremented and *dirty* field is updated using reference's *rw* field from the MRT. If buffer is mapped to any write-access reference, then write-access must be reflected in the *dirty* field to notify that buffer will be modified. In a case of a miss, some buffer in the local memory is selected to serve a miss and the Directory Block automatically schedules *dma_get* command to transfer mapped data.

The main difference compared to caches is that DB does not support replacement of a buffer. It is designed to maintain free and used buffers so that only free buffers can serve new references. The *counter* field is used for this purpose. Buffer is free if its *counter* field is equal to zero. If there are no free buffers, then no more references can be mapped to the local memory. This constraint is necessary hence all buffers assigned to references must be

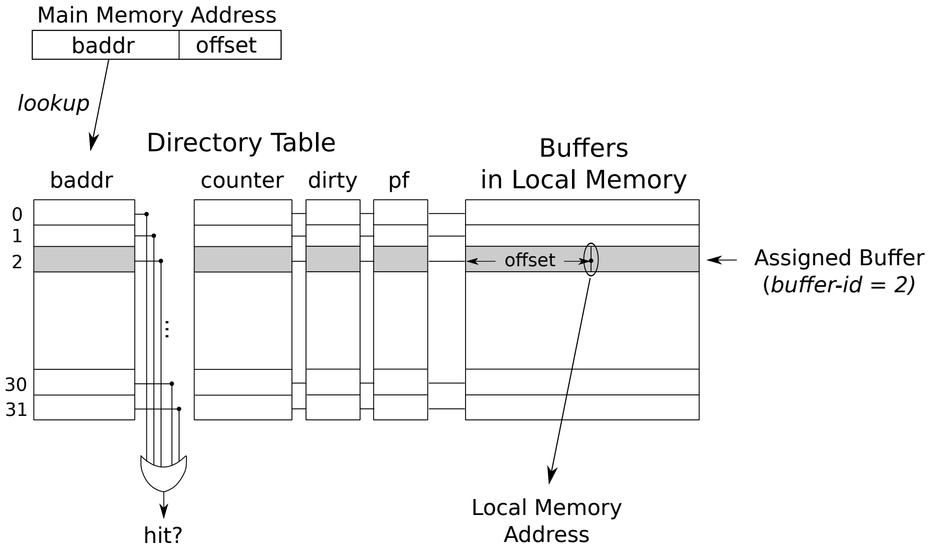


Figure 7.3: Directory Block.

resident in the local memory during the whole evolution of the actual work. It must not happen that one buffer get replaced by the other when both buffers are needed. So, new DMA controller can handle as many references as there are entries in the Directory Table. In this work, we consider 32 entries in the DT. Code using new DMA controller must be aware of this limitation and must not try to handle more than possible number of references. Limitation of 32 references seems not to be a big problem for scientific High-Performance Computing kernels since even our software solutions using basic DMA controller had similar limitations (Chapter 3). However, DMA-circular is aimed for acceleration purposes, so it should handle as many references as possible while others can be handled using some other, less efficient, techniques.

Besides incrementing the *counter* field, it is also necessary to decremented it when a reference does not use buffer any longer. Memory References Block is responsible for notifying Directory Block when to decrement the counter. If the *iters* field in the Memory References Table reaches zero because reference reached the end of the buffer then prior to triggering lookup in the DB to request new buffer, it is necessary first to release currently assigned buffer and this results in decremented *counter* field. Additionally, if the *counter* field of the dirty buffer reaches zero value then DB schedules *dma_put* command to transfer buffer back to the main memory.

#### 7.2.1.4 Prefetch Block

This block is responsible for prefetching action. It uses a Prefetch Queue that preserves the addresses to be used for prefetching. When Directory Block does lookup for an address then according to the sign of the stride of the reference, DB schedules prefetching of the next buffer by placing its base address in the Prefetch Queue. Once prefetching action is triggered by the user, Prefetch Block starts invoking lookup in the DB for all addresses found in the Prefetch Queue. User should invoke this operation before actual computation in order to overlap computation with communication. When prefetching buffers, DB sets *pf* field in the DT, but it does not increment counter field since prefetched buffers are not assigned to any reference.

#### 7.2.1.5 Translation Block

This component is an interface for the hardware to communicate to the software translated addresses resulting from the buffer assignment. When requesting a buffer for a reference, its main memory address is provided to the Directory Block which does buffer assignment and outputs the local memory address within the assigned buffer. These local memory addresses are preserved in the Translation Table so that a user can read them and use in the work phase.

#### 7.2.1.6 The reg-iters and reg-tags Registers

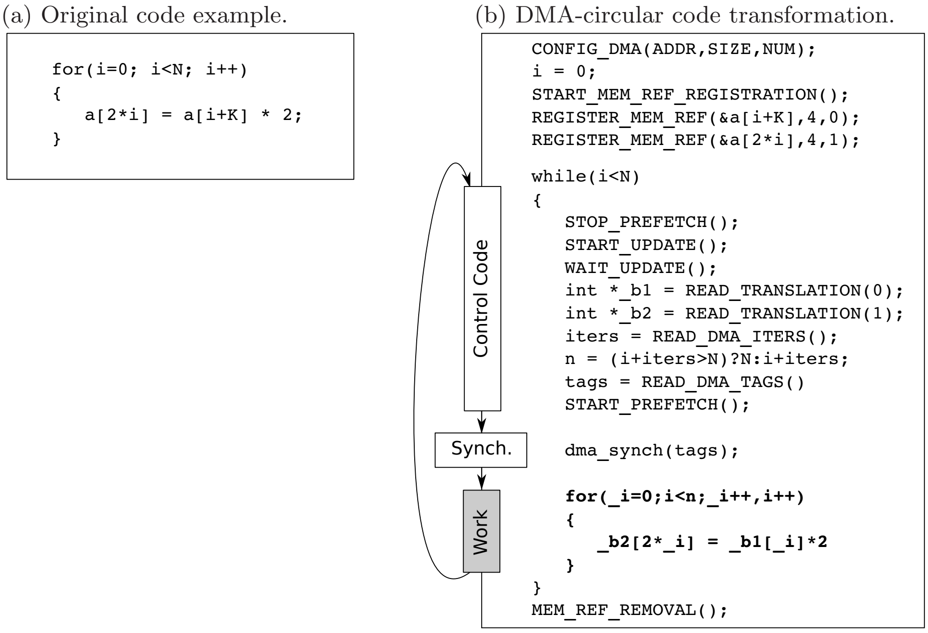
The reg-iters register is used to provide the software with the number of iterations that can be done in the next work phase. This register is managed by the Memory References Block.

The reg-tags register is used to provide the software with the tags to be used for the synchronization with all transfers related to the buffers that are needed in the work phase. Those transfers are either transfers resulting from the miss handling in the Directory Block or prefetching transfers in a case that some lookup hits in a prefetched buffer (buffer with the *pf* field set). The reg-tags register is managed by the Directory Block.

In the next section, we describe in more details how all these components work together.

### 7.2.2 Operational Model

In this section, we describe how the DMA-circular works and how to use it from the user point of view. In Figure 7.4, we present a code example (Figure 7.4(a)) and its code transformation to operate with the new DMA engine (Figure 7.4(b)). When operation with DMA-circular, the first thing to do is to configure the DMA-circular as described in Section 7.2.1.1. This is annotated with the CONFIG_DMA macro in Figure 7.4(b). Then the registration of references takes place. The macro START_MEM_REF_REGISTRATION annotates begining of the new loop which triggers actions in the Memory References Block. A value pointing to the last registered reference in MRT (total num of references in Figure 7.2) is pushed at the top of the Loop Stack (Figure 7.2) in order to remember the context of the MRT before references from the new loop are registered. Pointer to the top of the Loop Stack is actually pointer to the current nest level, as shown in Figure 7.2. Registration itself is done by REGISTER_MEM_REF macro which is responsible for placing descriptions of references in the Memory References Table. In our example, two references are registered (*a[i+K]* and *a[2*i]*). Figure 7.5(a) shows the MRT after the registration of these two references for their given footprint in the maim memory.



(a) Original code example.

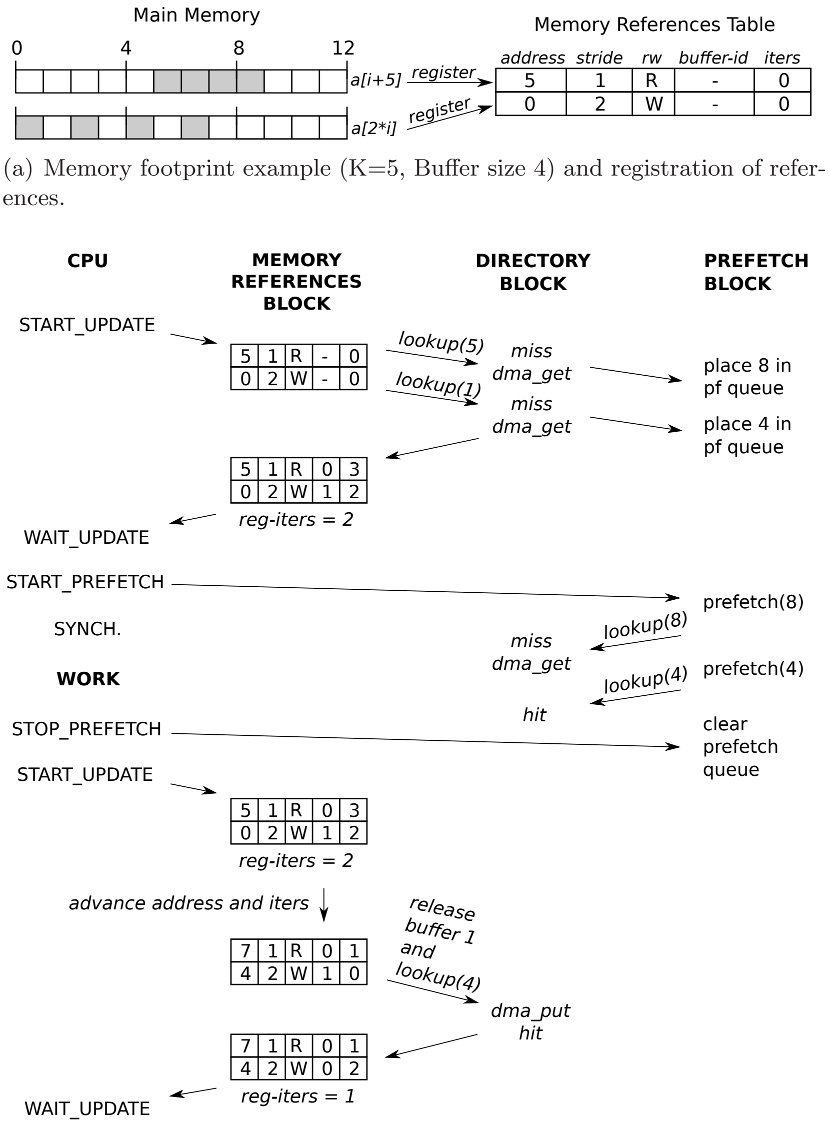
```
for(i=0; i<N; i++)
{
   a[2*i] = a[i+K] * 2;
}
```

(b) DMA-circular code transformation.

```
CONFIG_DMA(ADDR,SIZE,NUM);
i = 0;
START_MEM_REF_REGISTRATION();
REGISTER_MEM_REF(&a[i+K],4,0);
REGISTER_MEM_REF(&a[2*i],4,1);

while(i<N)
{
   STOP_PREFETCH();
   START_UPDATE();
   WAIT_UPDATE();
   int *_b1 = READ_TRANSLATION(0);
   int *_b2 = READ_TRANSLATION(1);
   iters = READ_DMA_ITERS();
   n = (i+iters>N)?N:i+iters;
   tags = READ_DMA_TAGS()
   START_PREFETCH();

   dma_synch(tags);

   for(_i=0;i<n;_i++,i++)
   {
      _b2[2*_i] = _b1[_i]*2
   }
}
MEM_REF_REMOVAL();
```

Figure 7.4: A code example that operates with DMA-circular.

The most important actions in the control code are START_UPDATE and WAIT_UPDATE. In Figure 7.5(b), we present flow diagram and evolution of Memory References Table during the first two executions of the control code for the new DMA controller. We can see that START_UPDATE will trigger actions in Memory References Block to bring buffers for references which *iters* field in the MRT is zero. Since, in the beginning, both references are without assigned buffer, then two lookups (lookup for address *5* and *1*) are triggered in Directory Block. Since there are no buffers in local memory, both lookups miss and DB schedules two *dma_get* commands to bring new data. Additionally, two addresses (*8* and *4*) are placed in the Prefetch Queue in order to be used for prefetching later. In return, DB updates *buffer-id* entries in MRT. We can see that one reference got buffer *0* assigned, and the other got buffer *1*. Along with this, MRB updates the *iters* fields. We can see that, for the given example in Figure 7.5(a), reference *a[i+K]* can do three iterations in the first buffer while the reference *a[2*i]* can do two iterations in the assigned buffer. Min of these two is placed in reg-iters, and it denotes the maximum number of iterations to be executed in the work phase. All this time while the MRB and DB are doing these actions CPU is waiting on WAIT_UPDATE. Once all updates are done, control is returned to CPU, and it can proceed with the next actions: reading local memory addresses from the Translation Table (macro READ_TRANSLATION in Figure 7.4), reading *reg-iters* register (macro READ_DMA_ITERS), adjusting iteration bound for the inner most loop, reading *reg-tags* register (macro READ_DMA_TAGS), starting of the prefetching (macro START_PREFETCH), synchronization with read tags, and finally actual work phase. Prefetching is started prior to synchronization, and in Figure 7.5(b) we can see that while CPU executes synchronization and work phases, Prefetch Block triggers lookup in



| address | stride | rw | buffer-id | iters |
|---|---|---|---|---|
| 5 | 1 | R | - | 0 |
| 0 | 2 | W | - | 0 |

(a) Memory footprint example (K=5, Buffer size 4) and registration of references.

(b) Flow diagram with the evolution of the MRT for the first two executions of the control code.

Figure 7.5: Operational Model.

DB for addresses found in Prefetch Queue. In our example, those addresses are *8* and *4*. Former will miss in the DB and *dma_get* will be programmed, while latter will hit.

After the work phase, control phase is repeated since not whole workload has been executed. In the second control phase, first action to be taken is to stop prefetching and start update in the MRB again. Now, MRB first updates *address* and *iters* fields. The *address* field is updated to the next address to be accessed ($address = address + stride \times reg - iters$), and the *iters* field is decremented by *reg-iters* number of iterations. We can see this evolution of MRT in Figure 7.5(b). Since now *iters* is zero for the *a[2*i]* reference, then prior to invoking lookup for the new address (address *4*) it is necessary first to release old buffer (buffer *1*). Since *a[2*i]* is write access reference then when releasing buffer *1*, DB schedules *dma_put* command to transfer modified buffer back to the main memory.

So, this process repeats until whole original iteration space for the loop is done. At the end when exiting from the loop, macro MEM_REF_REMOVAL (Figure 7.4) is used to release all buffers assigned to references from the current nesting level and to notify MRB that we are exiting from the loop so that Loop Stack (Figure 7.2) can be popped in order to switch the MRB to work with the references from the previous nesting level.

### 7.2.3 Applicability

New DMA engine conforms the best to distributed systems where each core work with private data (e.g., Intel SCC architecture). The reason for this applicability is the way how DMA-circular does write-back of the modified data. DMA-circular schedules *dma_put* commands to write whole buffer of data back even if only a portion of a buffer is modified. In Figure 7.5(b), *dma_put* is scheduled to transfer a buffer where actually every second element is modified. So, it can be used in shared memory systems for read-only references or even for write-access references when a user is sure that buffer is not going to be shared among cores.

## 7.3 Evaluation

The evaluation section is divided into five parts. First, we explain the methodology used in the evaluation. Second, we measure control code overheads with basic DMA controller. Third, we measure the effect of control code acceleration. Fourth, we study the overall impact in performance of the proposed DMA controller, and finally we study energy consumption.

### 7.3.1 Methodology

We have used loops from CG, IS, FT, and MG applications. In this evaluation, we execute them on a single-core. Obviously, we can execute parallel MPI version of these benchmarks and use DMA-circular on each node but the parallel execution is out of the context of this work. Our aim is to evaluate the effect of offloading the control code, for dealing with stride access references with constant strides, from the software to the new DMA controller. For this purpose, it is enough to use single core. We have selected mentioned loops because they are HPC workloads which contain plenty of stride access references with constant strides which are the only references to be addressed by the proposal in this chapter.

All loops are coded to use basic DMA controller and DMA-circular. Baseline is the implementation that uses basic DMA controller, and this implementation is based on our High Locality Cache (Chapter 3). To execute selected loops, we use the PTLsim simulation infrastructure, as described in Section 2.3.2. Additionally, we extend PTLsim with a local memory, a traditional DMA controller, and DMA-circular. In Table 7.1, we present the configuration parameters used for the simulations.

We configure PTLsim to work with a hybrid memory model where on-chip local memory is integrated in the core, side to the cache hierarchy. Figure 7.6 shows the high level design of the hybrid memory model used in this evaluation. Used hybrid memory model has been recently proposed in a study about local memory design space exploration for High-Performance Computing [15].

In order to operate with the integrated local memory, an address range of physical addresses is reserved and direct-mapped to logical addresses. When a memory operation is executed, a range check for the logical address is performed to determine if the memory operation is to the local memory or to the cache hierarchy. This check is done in parallel with the segmentation mechanism, prior to any action of the logical to physical address translation in the Memory Management Unit (MMU) [50]. If the virtual address is in the range reserved for the local memory the MMU is bypassed and a physical memory address that points to the local memory is generated [15]. Avoiding the address translation in MMU keeps the access time to the local memory fast and constant, which are two very

Table 7.1: 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<br>256 floating point registers |
| L1 I-cache | 32 KB, 8-way set-associative, 64-byte lines<br>2 cycles latency |
| L1 D-cache | 32 KB, 8-way set-associative, 64-byte lines<br>2 cycles latency |
| L2 cache | 256 KB, 24-way set-associative, 64-byte lines<br>15 cycles latency |
| L3 cache | 4 MB, 32-way set-associative, 64-byte lines<br>40 cycles latency |
| Local memory | size = 32 x studied buffer size<br>2 cycles latency |
| DMA-circular | 1 cycle latency: loop stack operations, accessing an entry in MRB, lookup in DT, prefetch in DT, releasing a buffer in DB, writing to Translation Table, and queuing an address in to Prefetch Queue<br>4 cycles latency to calculate iters in MRB, with maintaining the minimum iters value.<br>3 cycles latency to update address and iters fields in MRB with maintaining the minimum iters value. |
| I/O mapped load and store | 6 cycles latency |

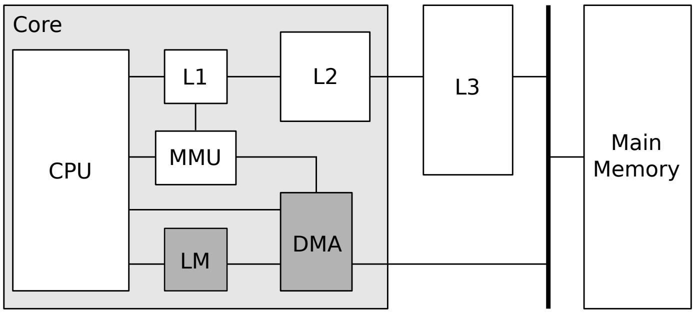


Figure 7.6: Simulated architecture with the hybrid memory model.

good properties of local memories. Address translation in the MMU is done only if the access is directed to the cache hierarchy.

As we introduced in the first chapter of the thesis, memory coherency at the DMA transfer level is important in the systems where on-chip local memories are mixed with caches. In the simulated system, it is ensured on the following way. 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 local memory, otherwise it is copied from the main memory. The bus requests generated by a *dma-put* copy the data from the local memory to the main memory and, in case the cache line is present in the L3 cache, it is invalidated.

Hybrid memory model is used in this evaluation for the simplicity reasons. We handle irregular memory references through the cache hierarchy only, while the local memory is exclusively devoted to the regular references (stride access references with constant strides). Simulated memory model, potentially, opens the system to be inconsistent since local memories are not coherent with caches. Therefore, it must not happen that the local memory and L1 cache work with the same data at the same time and that at least one of them modifies the data. We have checked and it is not happening in the simulated loops.

Good side is that a new problem is emphasized. As a result of some preliminary studies, we find that the presence of the Directory Block in our DMA-circular enlarges the possibilities of supporting the coherency of local memories and caches in the hybrid memory models. However, this study is ouf of the context of this thesis and we leave it for the future work.

### 7.3.2 Control Code Overheads

The main aim of this section is to analyse control code overheads when dealing with basic DMA controller, and determine an upper bound for the acceleration effect. For that purpose, all tested loops are described in terms of overheads. Figure 7.7 shows the time distribution for the selected loops in CG, FT, IS, and MG applications. Total execution time has been broken down to control, synchronization and work components. In Figure 7.7, percentages of the total execution time are displayed. Each loop is tested with different configuration for the size of the buffers. We used buffers of 1KB, 2KB, and 4KB. There are

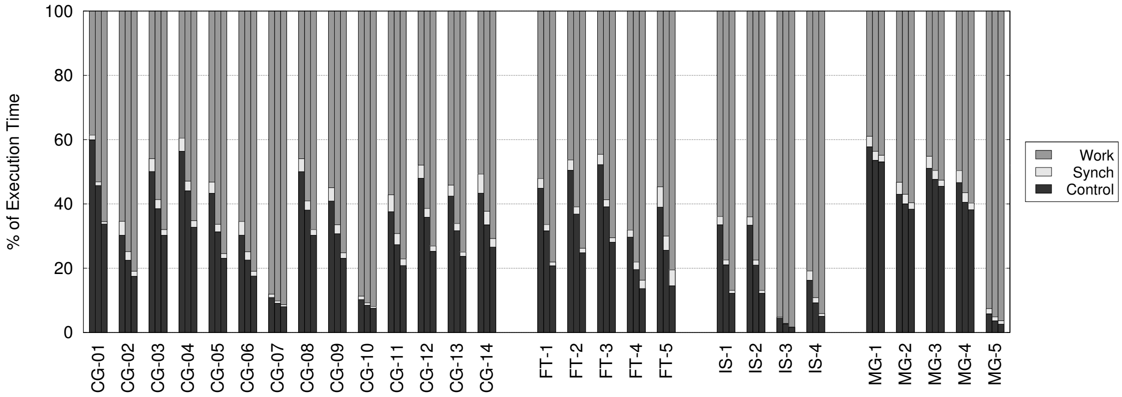


Figure 7.7: Control code overhead with basic DMA controller. Three bars associated to each loop are for three different buffer sizes. From left to right, sizes are: 1KB, 2KB, and 4KB.

three bars corresponding to each loop in Figure 7.7. The leftmost bar corresponds to buffer size of 1KB, middle bar is for buffers of 2KB in size, and the rightmost bar represents usage of 4KB buffers.

We can see in Figure 7.7 that a variety of overheads is present in the tested loops. There are loops with very little overhead, less than 20% of the execution time and loops with huge overheads of up to 60% of the execution time.

Loops which are not suffering from control code overheads are: CG-07, CG-10, IS-3, IS-4, and MG-5. These loops are computation bounded loops where very little of the execution time is spent in the control code due to very few references handled there. On example, loop 4 in IS application, handles only two references which are aliasing and sharing a buffer almost all the time. In other words, while the actual work performs operations for two references, control code is almost as efficient as control dealing with only one reference since handling one of the references ensures data for the other reference as well in this loop. Only 5% of the execution time is spent in the control code of IS-4 loop, when working with big (4KB) buffers. Big buffers require less frequent control code interventions for a given workload than smaller buffers. We can see that overhead increases when smaller buffers are in use. When 1KB buffer is used then overhead is around 20% of the execution time in this loop. This means that what DMA-circular is trying to improve, corresponds to 5%-20% of the total execution time in this case.

Loop 10 in CG application contains one scalar reduction variable and three references which perform array access. One of those three references is irregular and cannot be handled by DMA-circular. This reference is handled through cache hierarchy since it does not alias with any of the references mapped to the local memory. Irregular access pattern of this reference results in requiring missing data along the computation and then the work component takes the most significant part of the computation in this loop, while the control code, which handles the other two references, is at most 10% of the execution time. Similar situation is with the CG-07, IS-3 and MG-5 loops.

In the rest of CG loops we observed overheads ranging from 20% to 60% of the execution time. The case of CG-02, CG-05, CG-06, CG-09, and CG-13 are simple loops which do reduction on one or two arrays. Computation is fast and performed over, at most, two arrays and two scalar reduction variables. Due to the fast computation in these loops, control code takes a significant part of the execution time. Control code overhead is around 20% for big buffers, while it is close to 30% and 40% for 2KB and 1KB buffers, respectively. The rest of CG loops, as well as IS-1 and IS-2 loops, consist of simple computations over one or two arrays. Some of them does only copy operation or initialization of arrays, such as loop CG-01 which does initialization of six arrays. Due to a simple computation and a high number of references treated in the control code, loop CG-01 ends up in huge overheads ranging from 30% to 60% of the execution time.

The case of FT application consist of more complex kernels with plenty of loops and with more references to be handled than in IS and CG applications. In the case of FT-1, FT-2 and FT-3, 20% of the execution time corresponds to the control code when 4KB buffers are used. With 2KB buffers, observed overhead is around 35%, while with 1KB buffers, overhead is around 50% of the execution time. The FT-4 and FT-5 kernels threat less references than FT-1, FT-2, and FT-3 kernels and, as shown in Figure 7.7, result in less overhead which ranges from 15% to 40%.

The case of MG-1, MG-2, MG-3, and MG-4 are interesting examples where control overhead is high, around 40% or 50%, for all tested buffer sizes. Buffer size does not make any significant difference for the control overhead in these kernels. For example, MG-4 kernel (the most time consuming kernel in MG) is a case of 3 nested loops where the inner-most loop has short iteration space and considerable number of references (18 references handled in the control code). Due to short iteration space, the majority of iterations for the inner most loop can fit in only one 1KB buffer, which means that very few buffers (mostly one or two) per reference, can cover all inner-most iteration space. So, after the inner most loop is sub-chanked with very few buffers and executed, the outermost loops are executed. Afterwards, the control code for the new instance of the inner most loop is executed again. So, instead of having significant number of control code phases due to sub-chunking the inner-most loop, here, no matter the size of the buffer, number of control code instances in the inner-most loop is similar. Execution of the control code is dictated by the iteration space of the outer loops. Acceleration bound for the DMA-circular in this case is high, since 50% of the execution time could be addressed.

In conclusion, the potential effect of acceleration ranges between 5% and 50%, depending on the loop type and the buffer size. Loops which are computation bounded with very few references handled in the control code (for instance, CG-10 and IS-4), offer low potential for acceleration (between 5% and 20%, depending on the buffer size). Big overheads (from 20% to 50%) can be expected from variety of loops, such as: loops which are not computation bounded so that weight is pushed on the control code side (e.g., many CG loops), nested loops with short inner loops (e.g., MG-1, MG-2, MG-3, and MG-4 kernels), and any type of loop where the majority of references that appear in the work phase are mapped in the local memory and handled in the control code (e.g., FT kernels).

Next section evaluates acceleration effect of the DMA-circular.

### 7.3.3 Acceleration effect evaluation

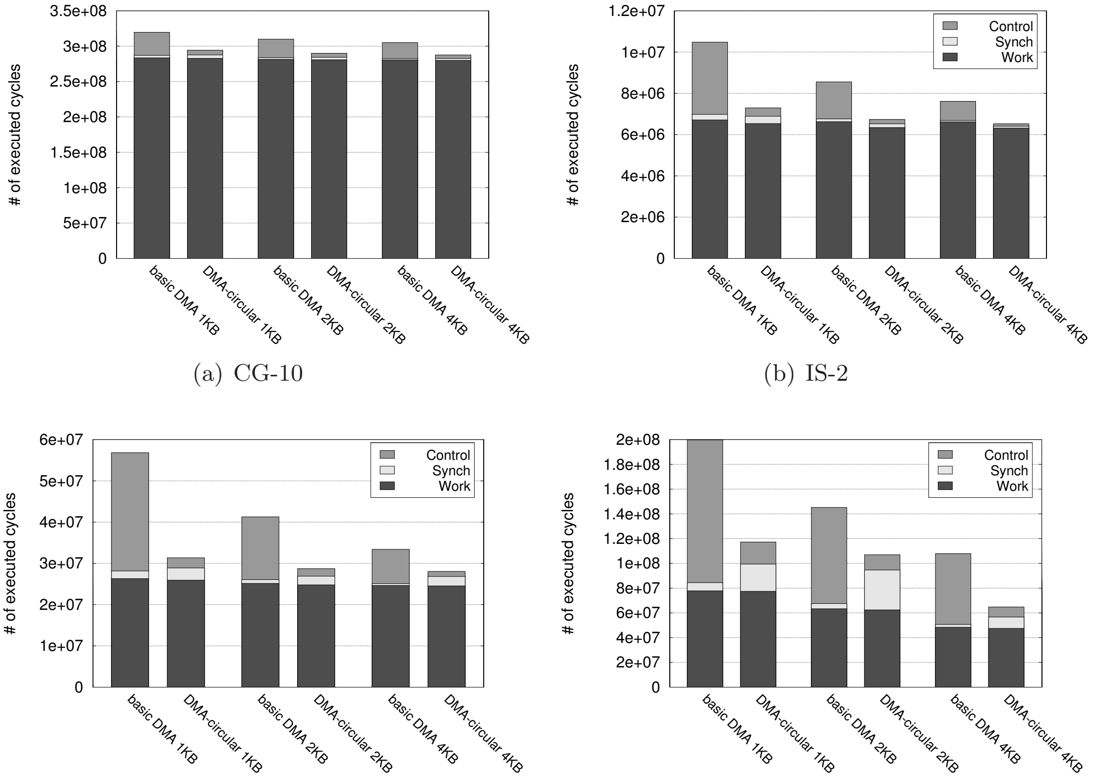
This section measures and checks the effectiveness of DMA-circular. We have select 4 loops (CG-10, FT-2, IS-2, and MG-1) as representative of what has been observed in the tested applications to explain the acceleration effect in detail. Figure 7.8 shows the execution of the selected loops, each one executed using basic DMA controller and DMA-circular. Execution time is decomposed into actual work, control overhead and synchronization. Each loop is executed using three different buffer sizes: 1KB, 2KB, and 4KB.

The case of CG loop 10, in Figure 7.8(a), represents a case dominated by the work component. Control code, in this loop, corresponds to less than 10% of the execution, but we can see that even that small control code has been significantly decreased with DMA-circular. New control code corresponds to just few percent of the execution time. Execution time of the work component remains flat in all versions which is expected since DMA-circular imposes changes in the control code only, while the source code for the work phase remains unchanged. Finally, the synchronization time gets a bit increased due to much shorter control code which is not long enough to completely overlap communication and computation when DMA-circular is used.

The case of loop 2 in IS application is a loop which is still dominated by the work component but with noticeable portion of the execution time devoted to the control code execution. Figure 7.8(b) shows the execution time breakdown for this loop and we can see that control code is around 30% when buffers of 1KB are in use, while it is a bit less for smaller buffers: 20% for 2KB buffers and 15 % for 4KB buffers. DMA-circular successfully accelerates original control code in this loop by decreasing it to very few percentage points. In a case of 1KB buffers, control code is accelerated almost 9 times, from its 33% percent in the baseline, it turns to only 5% percent of the execution time in the accelerated version. Similarly to the CG-10, synchronization is a bit increased here also. Additionally, we observed in this loop some slight variations in the execution time of the work component. Control code for the new DMA is much shorter which sometimes can make an impact on compiler to better optimize the work phase itself. We observed variations of at most 10% in the work components of all the tested loops.

A middle case in terms of control code dominance is FT loop 2 in Figure 7.8(c). In this loop, control code corresponds to 50%, 35% and 25% of the execution time for buffers of 1KB, 2KB, and 4KB respectively. The trend of increased synchronization due to accelerated control time, is clearly noticeable in this loop. In the case of 4KB buffers, original control code has been accelerated by factor of seven, while the synchronization has been increased by factor of five. Interestingly, if we calculate the same numbers but for smaller buffers, then we get control code acceleration by factors of 8 and 12 for 2KB and 1KB buffers, respectivelly. This means that acceleration effect of the DMA-circular is better for smaller buffers, and thus accelerated control code for smaller buffers is similar to the accelerated control code for bigger buffers, or at least, variations are not that drastical as they are when basic DMA is used.

At the end, the case of loop 1 in MG application (Figure 7.8(d)) is representative of loops dominated by the control code. In this loop, for all variations of buffer sizes, control code is more than 50% of the execution time when basic DMA is used. Acceleration of the control



(a) CG-10

(b) IS-2

(c) FT-2

(d) MG-1

Figure 7.8: Execution time decomposition into actual work, control overhead and synchronization for CG-10, IS-2, FT-2, and MG-1 loops, using basic DMA and DMA-circular with three different buffer sizes (1KB, 2KB, and 4KB).

code in this loop is significant and it is around factor of seven. Interesting observation, in this loop, is a drastic increment in the synchronization time, especially for 1KB and 2KB buffers, when DMA-circular is used. Synchronization time for 1KB and 2KB buffers is a bit bigger in this loop than in the other loops in Figure 7.8. This is because MG-1 suffers from wrong prefetching when 1KB and 2KB buffers are used while with 4KB buffers prefetching works fine. That is why in Figure 7.8(d), synchronization in a case of 1KB and 2KB buffers is much higher than in a case of 4KB buffers. However, increment in the synchronization time is not that drastic to diminish effects of the acceleration and thus speedups in total are observed. We can also see that the execution time of the work phase depends on the size of the buffer in MG loop 1, while in the other loops in Figure 7.8 it is not the case. This does not have anything with DMA-circular, this is an effect of subchunking the iteration space. It is faster to execute whole iteration space in one shot than to break it down in few shots. In the latter case, more jump instructions will be involved in total. For long iteration spaces (e.g.: CG-10, FT-2, and IS-2) increment of jump instructions has negligible impact on the total number of executed instructions but for short iteration spaces, such as MG-1, it has a significant impact which is noticeable in Figure 7.8(d).

In conclusion, we have observed similar trends in all tested loops. DMA-circular succeeds in diminishing the control code cycles without incurring an unacceptable increase in the other components of the execution time. Synchronization usually increases with DMA-circular since new control code is not long enough to overlap all transfers, but that increment is not big enough to diminish the effect of the acceleration in the total execution time of the loops. We have observed that speedup factors for the acceleration of the control code ranges from 6 to 12, usually being more efficient for smaller buffers than for bigger buffers in the tested loops.

Finally, notice that new code operating with small buffers is sometimes faster than baseline's code operating with big buffers. On example, in FT-2, IS-2 and CG-10 loops, total execution time with DMA-circular and 1KB buffers is better than the total execution time for the basic DMA and 4KB buffers. In a case of MG-1, it is not the case due to long synchronization when 1KB buffers are used with DMA-circular. But we can see that MG-1 with DMA-circular and 2KB buffers performs, in total, almost the same as basic DMA with 4KB buffers. This observation is interesting since it means that DMA-circular could save area by requiring less local memory in order to achieve comparable performance as the basic DMA with much bigger local memory. This is an interesting parameter for the study and in Section 7.3.5, we analyse what the trends are in the rest of the tested loops.

### 7.3.4 Overall loop performance

In this section, we evaluate the overall loop performance. We compare basic DMA against DMA-circular for three different buffer sizes (1KB, 2KB and 4KB). The comparison measures improvement in terms of speedup. Figure 7.9 shows speedup factors obtained in all the tested loops.

We saw that loops CG-07, CG-10, IS-3, IS-4, and MG-5 do not suffer from huge control code overheads. Accordingly, obtained speedup in those loops is low, less than 1.1 (Figure 7.9). In these loops we have very little space for optimization but good thing is that some speedups are obtained even in these loops which are not good candidates for acceleration.

Other loops, which had more significant control code overhead, benefit more from DMA-circular. On example, we saw in Figure 7.7 that FT loop 2 had control code overhead of 50%, 40%, and 20% for 1KB, 2KB and 4KB buffers, respectively. In Figure 7.9, we can see that obtained speedups in this loop for 1KB, 2KB, and 4KB buffers are: 1.8, 1.4, and 1.2 respectively. These speedup numbers correspond to the overheads seen in Figure 7.7. The same trend is obtained in the other loops. In general speedups are higher when small buffers are used. CG-01 reaches even two times faster execution with DMA-circular when small buffer of 1KB is used. This high speedup is obtained due to high control code overhead (around 60% of overhead in Figure 7.7) that is successfully accelerated with DMA-circular. Obtained speedups for 2KB and 4KB buffers are lower in majority of the tested loops. In CG, FT and IS loops speedup for 4KB buffers is around 1.2. In the MG loops, trend is a bit different. As we saw in Section 7.3.2, control code overhead for MG loops 1, 2, 3, and 4 are more or less the same no matter the buffer size. For the same reason, as expected, speedup is the same no matter the size of the buffers. We can see that MG-1, MG-2, MG-3,

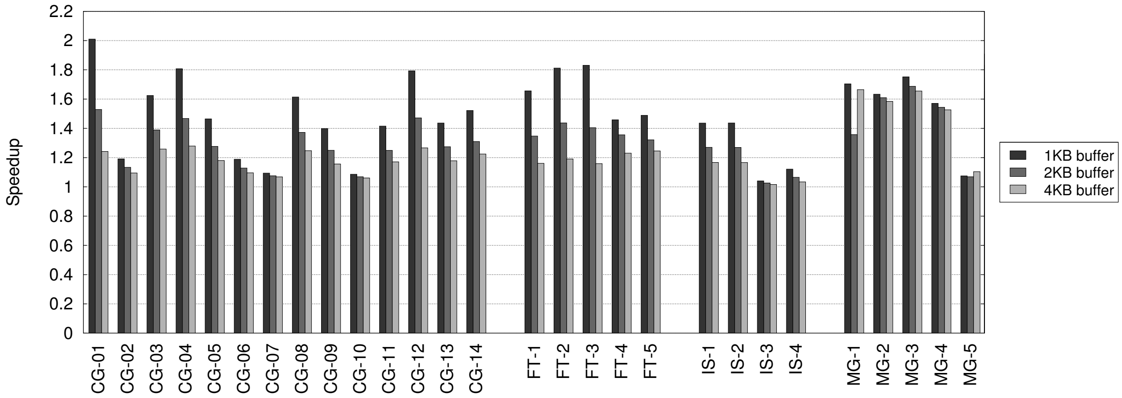


Figure 7.9: Speedup factors for the system using DMA-circular against system working with basic DMA. Three bars associated to each loop are speedups obtained for three different buffer sizes: 1KB (left-most bar), 2KB (middle bar), and 4KB (righ-most bar). Speedup is obtained by comparing execution time of the system working with DMA-circular and the selected buffer size against the system working with basic DMA and the same buffer size.

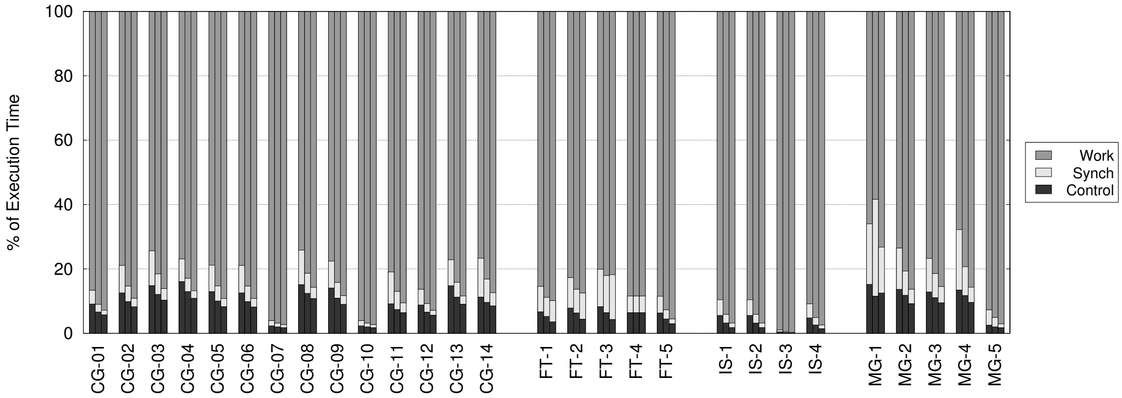


Figure 7.10: Control code overhead with DMA-circular. Three bars associated to each loop are for three different buffer sizes: 1 KB, 2 KB, and 4 KB (from left to right).

and MG-4 loops have speedups ranging between 1.4 and 1.5. Only MG loop 1 has a bit lower speedup for 2KB buffer and this is because of the increased synchronization time due to wrong prefetching, as explained in the previous section.

Finally, we accompany the speedup numbers with the Figure 7.10 which shows control code overhead when new DMA is used. In contrast to control code overhead with basic DMA (Figure 7.7, it is noticeable that new DMA reduces all control code overheads significantly. We can see that new control almost never exceeds 15% of the execution time while with the basic DMA it was going up to 60% of the execution time in some loops. On example, CG loop 4 had control code overhead of around 40% for 2KB buffers, while with DMA-circular

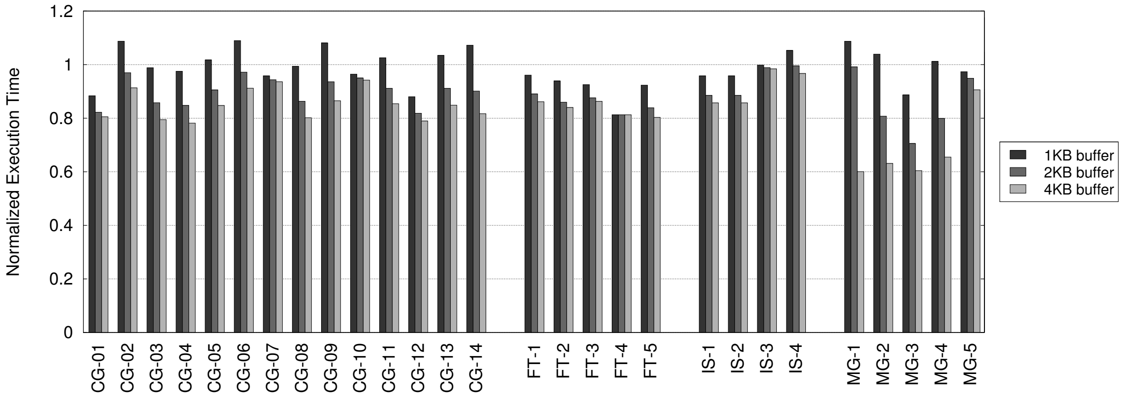


Figure 7.11: Execution time of all tested versions working with DMA-circular, normalized to the execution of the fastest version working with basic DMA (version working with 4KB buffers). Value one on Y-axis corresponds to the execution time of the version working with basic DMA and 4KB buffers.

it is around 15% which results in speedup of 1.4 for this particular loop and buffer size.

In general, obtained speedups ranges from 1.2 to 2 for 1KB buffers and from 1.1 to 1.6 for 2KB and 4KB buffers.

### 7.3.5 Impact on the Local Memory Size

In Section 7.3.3, we noticed that DMA-circular working with small buffers can be sometimes as efficient as basic DMA working with big buffers. We saw that this is a trend in CG-10, FT-2, and IS-2. In this section, we check what is a trend in the other tested loops.

In Figure 7.11, we compare the basic DMA working with 4KB buffers and the new DMA working with 1KB, 2KB, and 4KB buffers. The normalised execution time is present in Figure 7.11. The main observations is that the DMA-circular with a 1KB buffers almost always outperforms the basic DMA with 4KB buffers. In some loops, slowdown of up to 10% is observed. However, when DMA-circular works with 2K buffers then, in all tested loops, basic DMA that is using two times bigger buffers results in worse performance.

This is an important advantage of DMA-circular. Being able to achieve good performance with small buffers, what is not possible with basic DMA, has tremendous benefits, especially in area and power. In the basic DMA approach, the smaller the buffer, the more control bursts have to be executed so more overhead is introduced. This situation also happens with the DMA-circular, but with a big difference that the execution of the control code has a very low cost, as we saw in Section 7.3.3. and then the increment of control burst executions becomes affordable. Being able to efficiently work with small buffers translates to being able to have smaller local memories.

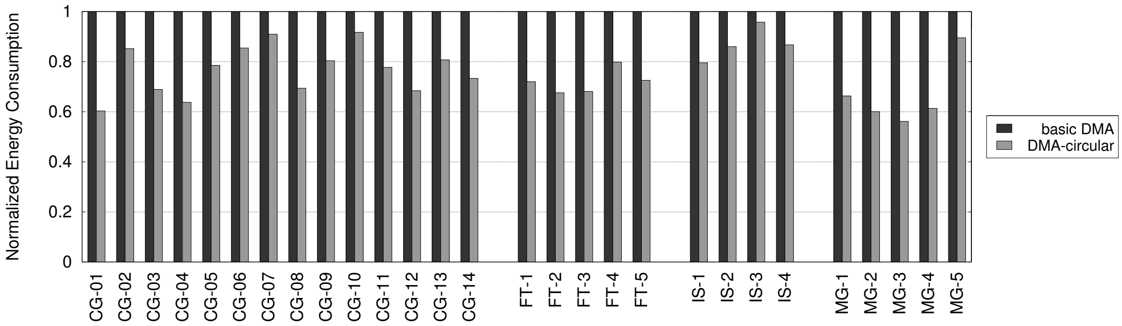


Figure 7.12: Energy consumption of the system working with DMA-circular, normalized to the energy consumption of the system working with basic DMA. Buffer size is fixed to 2KB.

### 7.3.6 Energy consumption and Power

Figure 7.12 compares energy consumption of the processor using basic DMA controller versus processor equiped with DMA-circular. Presented numbers are for a buffer size of 2KB.

We do not compare energy consumption of DMA controllers itself since we observed that energy consumed by the DMA engine itself is always less than 5% of the total energy consumption. In terms of energy the main source of benefit is the decreased instruction count in the control code. Less instructions executed in the control code decreases total energy consumed by a processor. For instance, in Figure 7.12 we observed that accelerated version of FT-1 benefits 30% in energy (see Figure 7.12) respect the baseline version. In this loop, observed speedup due to acceleration effect of DMA-circular (around 30% of the control code overhead has been decreased to 5% of overhead) is 1.3 and the same trend is observed for energy consumption. Since we observed speedups in all tested loops, we also observed more energy efficient execution of all loops. Figure 7.12 shows that energy savings for all the tested loops range from 5% to 40%. 5% of energy efficiency is observed in loops suffering from little speedups: CG-07, CG-10, IS-3, and MG-5. In the rest of the loops higher speedups are observed and then higher energy savings are present in Figure 7.12.

So, energy savings are obvious since shorter execution time ensures that but what is happening with power. We spend less energy in less time which requires power consumption study of the new design. In Figure 7.13, we compare power consumption of the processor working with basic DMA versus processor using DMA-circular. We can see that in all loops power consumption of the version using DMA-circular does not vary a lot from the baseline's power consumption. Observed variations in power are mostly within 10% of the baseline's power consumption.

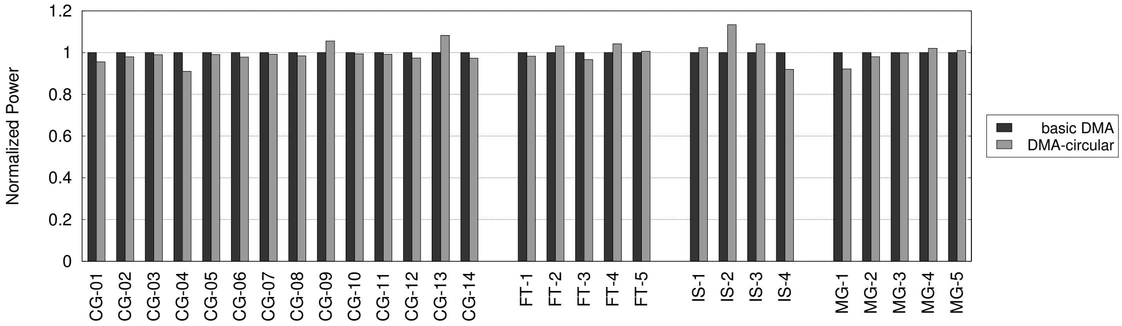


Figure 7.13: Power consumption of the system equiped with DMA-circular, normalized to the power consumption of the system working with basic DMA. Size of the used buffer is 2KB in this chart.

## 7.4 Related Work

Previous research works targeting the management of an on-chip local memory mainly differ on the existing hardware support for the actual management and how this hardware interacts with the software execution. In the case where no specific hardware support is available, explicit buffer allocation and a DMA programming style totally based on loop tiling techniques [21, 39] is the solution. This approach, however, requires precise information about memory aliasing of regular memory references at compile time which is not a case in our proposal.

S. Seo et al [98], propose a pure software cache architecture, with a configurable cache line size, a fast and adaptive placement and replacement mechanism for accelerator-based architectures with local memories. In general, this solution defines simple mechanisms and compiler code transformations to smooth the overhead related to the local memory management. The proposal in this chapter is a natural hardware design that accelerates several aspects of the software-based solutions. The proposed DMA engine is general enough to support and accelerate most of the mechanisms in this work.

Software caching techniques have been also applied to reduce the amount of power consumption associated to cache management. These proposals face similar problems as the ones addressed in this work and introduce explicit hardware support that could be adopted for the management of a local memory. For instance, Direct Addressed Caches [118] propose the elimination of the tag checks by making the hardware to remember the exact location of a cache line, so that hardware can access data directly. This proposal requires the definition of new registers in the architecture to relate load/store operation to specific cache lines, leaving to the compiler the decision of what memory references have to be associated to the additional registers.

In Osman Unsal et al. [111], a tag-less cache architecture for scratchpad memories is described. The miss handler is implemented in software but there is specific hardware to notify the program there has been a cache miss. New registers are introduced implementing the hotcachelines, associated to memory references that their access pattern can be easily predicted at compile time. The main aim of this approach is to reduce the energy consumption related to the tag checking. The scratchpad memory is totally managed by software, so there is no support for buffer allocation and memory disambiguation between buffers, and the eviction process of modified data is also under software control with no acceleration given by the hardware. All this points are addressed in the DMA-circular with dedicated hardware to accelerate these mechanisms.

Other works coming from embedded systems [25, 89] also address the management of a local memory. In general, these works target energy savings in the cache subsystem. The cache architecture is partitioned and fine grain placement mechanisms to skip the tag checks. The compiler generates enhanced load and stores operations that directly access to a specific partition. It is needed complex compiler analysis to appropriately map memory references to partitions. In general, this solution requires profile information, something very usual in embedded systems, but not so accepted for programming general purpose cores.

In this last context, similar ideas have been explored [42, 72, 93] but implemented on top of dynamic mechanisms supported in hardware at the cache logic level. These works define cache partitions to adapt to the observed locality at runtime. In general, all these works define two trends: first, place the control of the cache events at software level, and second, dynamically accommodate the cache mechanisms to the access patterns in the computation. The proposal in this chapter mixes both strategies, and in particular takes the first strategy up to a limit where the DMA-circular is mostly controlled by software.

## 7.5 Conclusions

This chapter describes the design and evaluation of the DMA-circular, a new DMA controller that operates with on-chip local memories. The motivation of the design is to accelerate control code overheads of managing buffers in the local memories while not imposing programmability issues. The two key features of the new design are: the introduction of cache functionalities within the DMA controller and the addition of specific hardware blocks that accelerate the common control actions and events associated to the operability of a local memory.

The DMA-circular is evaluated with several high performance computational kernels of the NAS benchmark suite. The results indicate that the design reduces the control code overheads under 15% of the execution time. The overall performance of the tested kernels is improved between 1.2x to 2x when DMA-circular is used. Also, energy consumption is reduced from 5% to 40% respect the energy consumption when basic DMA is used. Reduced energy consumption is proportional to the obtained improvement in the execution time which turns the power disipation to be comparable to the power disipation of the systems using basic DMA

# Chapter 8

# Conclusions and Future Work

In this thesis, we address efficient usage and programmability of local memories in the context of both single core and multi-core processors. We start with software optimizations regarding the utilization of local memories and in that context, we develop an optimized infrastructure for software caching.

Out first contribution is motivated by overheads imposed by traditional software cache. As a result, we propose the novel hybrid access-specific software cache architecture that maps memory accesses according to the locality they expose. It presents a design of two distinct and custom cache structures that are tailored for two different kind of references: high-spatial locality and irregular references. Performance evaluation indicates that the hybrid design is significantly more efficient than traditional software cache approaches for a set of NAS parallel benchmarks, with speedup factors in the 2.8 to 8.5 range.

Afterwards, we continue optimizing our hybrid software cache. We analyze overhead distribution of our hybrid software cache and we locate new sources of overhead. The write-back of dirty cache lines imposes huge overhead in the initial implementation and we address that overhead for optimization. That is how we end up with the novel adaptive and speculative memory consistency mechanisms for our software cache. The main achievement is that of minimizing the impact on performance for the most critical aspects in software-based implementations: atomicity and control code execution related to dirty-bits data structures. Performance evaluation indicates that the proposed mechanisms are significantly more efficient: speedup factors between 20% and 40% are observed in several applications in the scientific domain.

We check again the overhead distribution or our software cache and we analyze the evolution from the traditional software cache to the hybrid software cache. As a result of the mentioned analysis, we locate new overhead to be addressed. This time it is about overlapping communication with computation. In that context, we develop two prefetching techniques for our hybrid access-specific software cache. We develop automatic prefetch for high-locality references and modulo scheduling transformation for irregular references. We observe that automatic prefetch can improve some loops by 20% due to better overlapping of the communication with the computation, while modulo scheduling transformation for irregular references results in speedups ranging from 20% to 40%.

So, at the end of the day we get optimized software cache infrastructure for efficient utilization of local memories. But this is not all. We continue analysing other problems related to the utilization of local memories and we propose optimizations on the hardware level in order to make handling of local memories more convenient and faster. One of those optimizations come up from the experience collected during the development of the software cache. We notice that DMA engines, responsible for data transfers to/from local memories, impose hard alignment constraints which prevent programmers from achieving the most convenient data allocation in local memories. Simply, allocation of data in local memory must resemble data alignment of the corresponding data from the global memory and this can make an impact on the efficient usage of SIMD units which also impose alignment constraints. Therefore, we describe the hardware proposal for dealing with unaligned memory transfers in modern high-performance architectures with local memories and DMA engines. We take advantage of the DMA transfers needed in the processors with local memories, and design the Realignment Unit that performs data realignment in the DMA engines while data is in transfer between local and global memory. With this support, programmers can move data from any alignment in the global memory to the most convenient alignment in the local memory and vice versa. This hardware offers a freedom of arranging the data in the local memory. Evaluation shows that the proposed Realignment Unit can be introduced in the DMA engines with no performance loss in terms of available bandwidth.

Finally, we propose to accelerate software cache handlers by mapping them to hardware. More precisely, we propose the design of the new DMA controller, namely DMA-circular, that is capable of accelerating control code overheads of managing buffers in the local memories while not imposing programmability issues. The two key features of the DMA-circular are: the introduction of cache functionalities within the DMA controller and the addition of specific hardware blocks that accelerate the common control actions and events associated to the operability of the local memory. The evaluation results indicate that the design reduces the control code overheads from 50% of the execution time under 15% of the execution time. The overall performance of the tested kernels is improved between 1.2x to 2x when DMA-circular is used. Also, energy consumption is reduced from 5% to 40% respect the energy consumption when basic DMA is used.

Last proposal motivate some future work regarding the coherency in the hybrid memory models where caches and local memories coexist in a processor. We know that local memories are not coherent since programmers are responsible for managing local memories and for maintaining the coherency with other entities in the memory hierarchy. However, caching functionalities introduced in the DMA-circular motivate us to think about hardware support and compilers' interaction for the coherency of local memories and caches in the hybrid memory models. We have observed encouraging preliminary results that we are going to address in the future work.

# Appendix A Publications

As the result of the research done so far, next papers have been published:

**Journal papers:**

- Nikola Vujic, Felipe Cabarcas, Marc Gonzalez, Alex Ramirez, Xavier Martorell, Eduard Ayguade, *"DMA++: On the Fly Data Realignment for On-Chip Memories"*, IEEE Transactions on Computers, vol. 61, no. 2, pp. 237–250, February 2012.

- Nikola Vujic, Marc Gonzalez, Xavier Martorell, Eduard Ayguade, *"Automatic Prefetch and Modulo Scheduling Transformations for the Cell BE Architecture"*, IEEE Transactions on Parallel and Distributed Systems (TPDS), vol. 21, no. 4, pp. 494–505, April 2010.

**Conference papers:**

- Nikola Vujic, Lluc Alvarez, Marc Gonzalez, Xavier Martorell, Eduard Ayguade, *"DMA-circular: an Enhanced High Level Programmable DMA Controller for Optimized Management of On-chip Local Memories "*, In Proceedings of the 9th ACM International Conference on Computing Frontiers (CF'12), Cagliari, Italy, May 2012. [Acceptance rate: 28%].

- Nikola Vujic, Marc Gonzalez, Felipe Cabarcas, Alex Ramirez, Xavier Martorell, Eduard Ayguade, *"DMA++: On the Fly Data Realignment for On-Chip Memories"*, In Proceedings of the 16th International Symposium on High-Performance Computer Architecture (HPCA-16), Bangalore, India, January 2010. [Acceptance rate: 18%].

- Marc Gonzalez, Nikola Vujic, Xavier Martorell, Eduard Ayguade, Alexander Eichenberger, Kathryn O'Brien, Kevin O'brien, Chen Tong, Zehra Sura, Tao Zhang, *"Hybrid Access-Specific Software Cache Techniques for the Cell BE Architecture"*, In Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT 2008), Toronto, Canada, October, 2008. [Acceptance rate: 19%].

**Workshop papers:**

- Nikola Vujic, Lluc Alvarez, Marc Gonzalez, Xavier Martorell, Eduard Ayguade, *"Adaptive and Speculative Memory Consistency Support for Multi-core Architectures with On-chip Local Memories"*, In Proceedings of the 22nd Annual workshop on Languages and Compilers for Parallel Computing (LCPC 2009), Newark, Delaware, US, October, 2009. [Acceptance rate: 58%].

- Nikola Vujic, Marc Gonzalez, Xavier Martorell, Eduard Ayguade, “Automatic Pre-Fetch and Modulo Scheduling Transformations for the Cell BE Architecture”, In Proceedings of the 21st Annual workshop on Languages and Compilers for Parallel Computing (LCPC 2008), Edmonton, Canada, August, 2008. [Acceptance rate: 51%].

# Appendix B Cell Processor

The Cell processor is the first-generation Cell Broadband Engine Architecture (CBEA) processor. It is a heterogeneous, multicore chip that consists of a PowerPC Processor Element (PPE), eight Synergistic Processor Elements (SPEs), high-bandwidth memory controller, and a high bandwidth I/O controller. All elements are connected via on-chip high-bandwidth Element Interconnect Bus (EIB). All key elements of the this processor and bandwidths between them are shown in Figure 1. In general, this processor is designed to work on a high frequencies but to be power efficient. The processor units operate on frequency of 3.2 GHz while EIB operates on a half of that frequency (1.6 GHz).

The PPE is a dual-issue, in-order 64-bit Power Architecture core with two-way simultaneous multithreading. This core has vector multimedia extension (VMS) unit, and a traditional cache hierarchy: 32KB L1 instruction and data caches, and a coherent 512KB L2 cache. The PPE is Cell's main processor that is aimed to run the operating system and schedule jobs on SPEs.

The SPEs are accelerator based cores which provide the most of the compute power in this system. Figure 2 shows the SPE's main components and bandwidths between them.

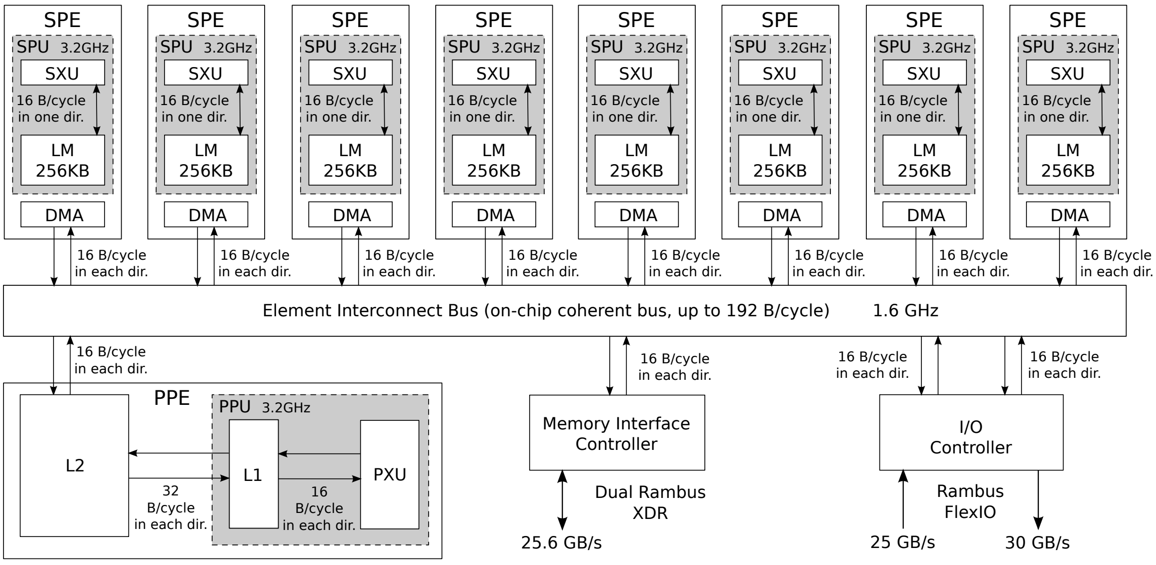


Figure 1: Block diagram of the Cell processor.

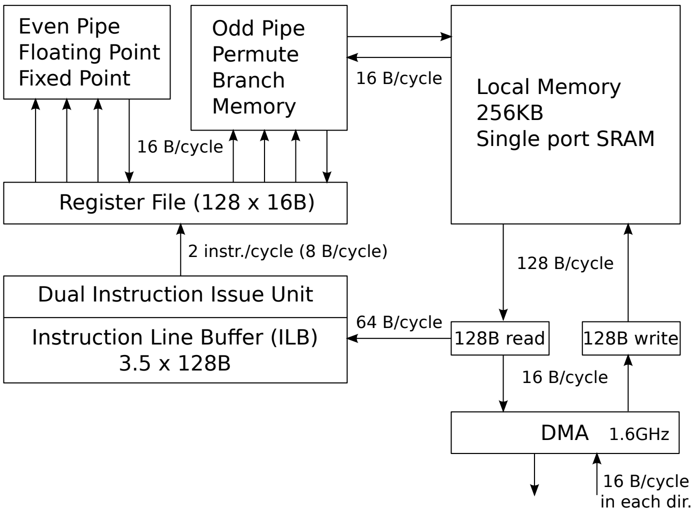


Figure 2: Block diagram of the SPE.

The SPE consists of a Synergistic Processor Unit (SPU) and a DMA engine. The SPU is a RISC-style in-order processor designed for high-performance data-intensive computations. All functional units in SPU are organized as Single Instruction Multiple Data (SIMD) units. The SPU has no cache hierarchy, it is provided with on-chip local memory shared for both data and program. The SPU cannot access main memory directly, but it can issue, under software control, a command to the DMA engine which is capable of transferring data between local memory and main memory.

Local memory is the largest component of the SPU. It is a single ported, 256 KB SRAM [33] that supports fully-pipelined 16-byte accesses for memory instructions and 128-byte accesses for instruction fetch and DMA transfers. Since SPU's local memory is single ported, then functional units, instruction issue unit, and DMA engine compete for the same port when accessing the local memory.

Instruction fetch occurs when local memory is idle. Instructions are fetched into Instruction Line Buffer (ILB) which is organized into 3.5 lines of 128 bytes. Each line is fetched in two cycles and can hold 32 instructions. All instructions in SPUs are 4-byte SIMD instructions which work with 16-byte of data at four granularities: 16-way 8-bit integers, eight-way 16-bit integers, four-way 32-bit integers or single-precision floating point numbers, and two 64-bit floating point numbers. The SPU can dispatch up to two instructions per cycle to execution units that are organized in two instruction pipes: *even* and *odd* pipe. Even pipe is only for floating and fixed point execution units, while odd pipe executes permute, branch and memory instructions. Each instruction is issued in-order and routed to a corresponding pipe. Table 1 shows instruction latencies and pipe assignments. Instructions in each pipe can operate on up to three 16-byte source operands and produce one 16-byte result. Thus, the unified register file supports six reads and two writes of 16 bytes per cycle (bandwidth of 51.2 GB/s due to frequency of 3.2 GHz) to one of its 128 16-byte registers. The memory instructions transfer 16 bytes of data between the register file and the local

Table 1: Instructions latencies and pipe assignment for SPE.

| Instructions | Pipe | Latency |
|---|---|---|
| Arithmetic, logical, compare, select | even | 2 |
| Shift, rotate, byte sum/diff/avg | even | 4 |
| float | even | 6 |
| 16-bit integer multiply-accumulate | even | 7 |
| 128-bit shift/rotate, shuffle, estimate | odd | 4 |
| Load, store, channel read/write | odd | 6 |
| Branch | odd | 1-18 |

memory. Supported bandwidth between memory unit and the local memory is 16 bytes per cycle, or 51.2 GB/s. When accessing local memory, some additional constraints are imposed. Each memory access must be aligned on a 16-byte boundary. In the architecture, it is ensured by always ignoring 4 lower bits of load/store byte address. In order to support load/store of values smaller than 16 bytes, instruction for extracting/merging an individual value from/into a 16-byte register is supported.

Execution of branch instructions is expensive in SPUs since there is no branch predictor. Branches are assumed not-taken but the architecture allows for a branch hint instruction. In order to have efficient execution of a correctly-hinted taken branches, the branch hint instruction prefetches, in one of the lines of the ILB, up to 32 instructions starting from the branch target. Also, a bit-wise select instruction is provided in the architecture in order to support elimination of short branches.

The DMA engine accesses local memory at granularity of 128 bytes and can support up to 16 DMA transfers at a time. Maximal supported DMA transfer size is 16KB and DMA transfers are issued on the bus in requests of 128 bytes. DMA transfers can be initiated by a program running on the SPE itself, by a program running on some other SPE, or by a program running on the PPE. The DMA engine can do transfers between SPE and any other element connected via EIB. In order to use correct physical addresses when issuing DMA requests on the EIB, the DMA engine translates all DMA requests by an MMU unit. The next three types of DMA transfers can happen:

- **SPE to SPE**. These transfers have a peak bandwidth of 25.6 GB/s since it is the bandwidth between DMA engine and EIB (16 bytes per cycle under EIB's frequency of 1.6 GHz).

- **SPE to main memory**. The aggregate bandwidth of these transfers for the entire Cell processor is 25.6 GB/s, since it is a bandwidth supported by the memory controller.

- **SPE to an I/O device**. A bandwidth of these transfers is conditioned by the bandwidth supported by the I/O controller. The I/O controller is configured to provide a peak bandwidth of 35 GB/s outbound, and 25 GB/s inbound.

Every link to the EIB has a bandwidth of 25.6 GB/s in each direction. I/O controller has two links to the EIB, while all the other elements are connected to the EIB via a single link. Having 12 (eight for SPEs, one for PPE, one for memory controller, and two for I/O controller) links connected to EIB, each one capable of sending and receiving 16 bytes in one EIB's cycle (25.6 GB/s), gives us that the EIB can transfer up to 192 bytes per EIB's cycle. Unfortunately, the rate of 192 bytes per cycle does not determine the EIB's maximum data bandwidth. The EIB's maximum data bandwidth is conditioned by the coherency mechanism and the rate at which addresses are snooped across the system. The mentioned rate is one address per EIB's cycle, where each address request can transfer up to 128 bytes. So, the EIB's theoretical peak data bandwidth is 204.8 GB/s. Finally, the EIB itself is organized as a ring bus of four 16-byte-wide data rings: two running clockwise and the other two counterclockwise.

For more details about the architecture and implementation of the Cell processor, we suggest reading of:

- the first paper about the Cell processor [52],
- its architecture overview with some details [44, 45, 56, 59], and
- the circuits and physical implementation [33, 84, 85, 107].

# References

[1] *IBM BladeCenter JS21*, 2006. URL ftp://ftp.software.ibm.com/common/ssi/pm/sp/n/bld01823us 11, 19

[2] *AMD Opteron 6000 Series Platform Quick Reference Guide*, 2011. URL http://www.amd.com/es/Documents/Opteron_6000_QRG.pdf. 4

[3] *Intel Xeon Processor E7-4870*, 2011. URL http://ark.intel.com/products/53579. 4

[4] TW Ainsworth, TM Pinkston, N.G.S. Technol, and R. Beach. Characterizing the Cell EIB On-chip Network. *IEEE Micro*, 27(5):6–14, 2007. 93

[5] P. Altevogt et al. IBM BladeCenter QS21 Hardware Performance. *IBM Technical White Paper WP101245*, 2008. 18

[6] M. Alvarez, E. Salami, A. Ramirez, and M. Valero. Performance Impact of Unaligned Memory Operations in SIMD Extensions for Video Codec Applications. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems & Software*, pages 62–71, 2007. 79, 82, 83

[7] AXI AMBA. Protocol specification v1. 0, 2003. 81

[8] C. Amza, A.L. Cox, S. Dwarkadas, L.I.J.I.E. Jin, K. Rajamani, and W. Zwaenepoel. Adaptive Protocols for Software Distributed Shared Memory. *Proceedings of the IEEE*, 87(3):467, 1999. 58

[9] Cristiana Amza, Alan L. Cox, Willy Zwaenepoel, and Sandhya Dwarkadas. Software dsm protocols that adapt between single writer and multiple writer. In *Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture*, HPCA '97. IEEE Computer Society, 1997. 58

[10] Murali Annavaram, Jignesh M. Patel, and Edward S. Davidson. Data prefetching by dependence graph precomputation. In *Proceedings of the 28th annual international symposium on Computer architecture*, pages 52–61. ACM, 2001. 76

[11] OCP-IP Association. Open core protocol specications v1.0, 2001. 81

[12] DH Bailey, E. Barszcz, JT Barton, DS Browning, RL Carter, L. Dagum, RA Fatoohi, PO Frederickson, TA Lasinski, RS Schreiber, et al. The Nas Parallel Benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63, 1991. 11, 17

[13] R. Banakar, S. Steinke, B.S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *Proceedings of the tenth international symposium on Hardware/software codesign*, pages 73–78. ACM, 2002. 5

[14] K.W. Batcher and R.A. Walker. Interrupt Triggered Software Prefetching for Embedded CPU Instruction Cache. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 91–102. IEEE Computer Society Washington, DC, USA, 2006. 76

[15] R. Bertran, M. Gonzàlez, X. Martorell, N. Navarro, and E. Ayguadé. Local memory design space exploration for high-performance computing. *The Computer Journal*, 54 (5):786, 2011. 5, 115

[16] D. Boggs, A. Baktha, J. Hawkins, D.T. Marr, J.A. Miller, P. Roussel, R. Singhal, B. Toll, and KS Venkatraman. The Microarchitecture of the Intel Pentium 4 Processor on 90nm Technology. *Intel Technology Journal*, 8(1):1–17, 2004. 80, 102

[17] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th annual international symposium on Computer architecture*, pages 83–94. ACM, 2000. 21

[18] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38–53, 2009. 18

[19] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Techniques for reducing consistency-related communication in distributed shared-memory systems. *ACM Transactions on Computer Systems (TOCS)*, 13(3):205–243, August 1995. 58

[20] T.F. Chen. An effective programmable prefetch engine for on-chip caches. In *Proceedings of the 28th annual international symposium on Microarchitecture*, pages 237–242. IEEE Computer Society Press Los Alamitos, CA, USA, 1995. 76

[21] Tong Chen, Zehra Sura, Kathryn O'Brien, and John K. O'Brien. Optimizing the use of static buffers for dma on a cell chip. In *Proceedings of the 19th international conference on Languages and compilers for parallel computing*, pages 314–329, 2007. 41, 125

[22] Tong Chen, Haibo Lin, and Tao Zhang. Orchestrating data transfer for the cell/b.e. processor. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 289–298, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-158-3. doi: http://doi.acm.org/10.1145/1375527.1375570. 41, 76

[23] Tong Chen, Tao Zhang, Zehra Sura, and Mar Gonzales Tallada. Prefetching irregular references for software cache on cell. In *Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*, pages 155–164, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-978-4. doi: http://doi.acm.org/10.1145/1356058.1356079. 41, 76

[24] G. Cheong and M. Lam. An Optimizer for Multimedia Instruction Sets. *Contract*, 30602:95–C. 102

[25] H. Cho et al. Dynamic data scratchpad memory management for a memory subsystem with an MMU. In *Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 195–206, 2007. 126

[26] Jamison D. Collins, Dean M. Tullsen, Hong Wang, and John P. Shen. Dynamic speculative precomputation. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 306–317. IEEE Computer Society, 2001. 76

[27] Jamison D. Collins, Hong Wang, Dean M. Tullsen, Christopher Hughes, Yong-Fong Lee, Dan Lavery, and John P. Shen. Speculative precomputation: long-range prefetching of delinquent loads. In *Proceedings of the 28th annual international symposium on Computer architecture*, pages 14–25. ACM, 2001. 76

[28] TM Conte, PK Dubey, MD Jennings, RB Lee, A. Peleg, S. Rathnam, M. Schlansker, P. Song, and A. Wolfe. Challenges to Combining General-purpose and Multimedia Processors. *Computer*, 30(12):33–37, 1997. 79

[29] JJ Costa, T. Cortes, X. Martorell, E. Ayguade, and J. Labarta. Running OpenMP applications efficiently on an everything-shared SDSM. *Journal of Parallel and Distributed Computing*, 66(5):647–658, 2006. 59

[30] *VAST-F/AltiVec: Automatic Fortran Vectorizer for PowerPC Vector Unit*. Crescent Bay Software. 102

[31] Raja Das, Joel H. Saltz, and Reinhard von Hanxleden. Slicing analysis and indirect accesses to distributed arrays. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 152–168. Springer-Verlag, 1994. 76

[32] M. Dasygenis, E. Brockmeyer, B. Durinck, F. Catthoor, D. Soudris, and A. Thanailakis. A combined DMA and application-specific prefetching approach for tackling the memory latency bottleneck. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 14(3):279–291, 2006. 76

[33] S.H. Dhong, O. Takahashi, M. White, T. Asano, T. Nakazato, J. Silberman, A. Kawasumi, and H. Yoshihara. A 4.8 ghz fully pipelined embedded sram in the streaming processor of a cell processor. In *IEEE International Solid-State Circuits Conference, Digest of Technical Papers.*, pages 486–612. IEEE, 2005. 132, 134

[34] K. Diefendorff, P.K. Dubey, R. Hochsprung, and H. Scales. AltiVec Extension to PowerPC Accelerates Media Processing. *IEEE Micro*, 20(2):85–95, 2000. 102

[35] S. Dwarkadas, K. Gharachorloo, L. Kontothanassis, D.J. Scales, M.L. Scott, and R. Stets. Comparative Evaluation of Fine-and Coarse-Grain Approaches for Software Distributed Shared Memory. In *Proceedings of the Fifth International Symposium on High-Performance Computer Architecture*, pages 260–269, 1999. 58

[36] Sandhya Dwarkadas, Alan L. Cox, and Willy Zwaenepoel. An integrated compile-time/run-time software distributed shared memory system. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, ASPLOS-VII, pages 186–197. ACM, 1996. 59

[37] 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 enginetm architecture. *IBM System Journal*, 45(1):59–84, 2006. ISSN 0018-8670. 10, 18, 41, 75

[38] A.E. Eichenberger, P. Wu, and K. O'Brien. Vectorization for SIMD Architectures With Alignment Constraints. In *Proceedings of the SIGPLAN Conference on Programming Languages Design and Implementation*, pages 82–93, June 2004. 81, 82, 102

[39] Alexandre E. Eichenberger, Kathryn O'Brien, Kevin O'Brien, Peng Wu, Tong Chen, Peter H. Oden, Daniel A. Prener, Janice C. Shepherd, Byoungro So, Zehra Sura, Amy Wang, Tao Zhang, Peng Zhao, and Michael Gschwind. Optimizing compiler for the cell processor. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 161–172, 2005. 81, 125

[40] J.B. Fryman, H.H.S. Lee, C.M. Huneycutt, N.F. Farooqui, K.M. Mackenzie, and D.E. Schimmel. SoftCache: A Technique for Power and Area Reduction in Embedded Systems. 42

[41] P.N. Glaskowsky. Nvidias fermi: The first complete gpu computing architecture. *NVIDIA Corp*, 2009. 5

[42] A. Gonzalez et al. A data cache with multiple caching strategies tuned to different types of locality. In *Proceedings of the 9th international conference on Supercomputing*, pages 338–347, 1995. 126

[43] M. Gschwind. Optimizing data sharing and address translation for the cell be heterogeneous chip multiprocessor. In *Computer Design, 2008. ICCD 2008. IEEE International Conference on*, pages 478–485. IEEE. 7

[44] M. Gschwind, P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. A novel simd architecture for the cell heterogeneous chip-multiprocessor. In *Proc. 2005 Symp. High-Performance Chips (HOT CHIPS 17)*, 2005. 18, 134

[45] Michael Gschwind, H. Peter Hofstee, Brian Flachs, Martin Hopkins, Yukio Watanabe, and Takeshi Yamazaki. Synergistic processing in cell's multicore architecture. *IEEE Micro*, 26(2):10–24, 2006. 31, 81, 134

[46] Michael Gschwind, David Erb, Sid Manning, and Mark Nutter. An open source environment for cell broadband engine system software. *Computer*, 40(6):37–47, 2007. 81

[47] Daniel Hackenber. *Fast Matrix Multiplication on Cell (smp) Systems*, 2007. URL `http://www.tu-dresden.de/zih/cell/matmul`. 17

[48] Luddy Harrison and Sharad Mehrotra. A data prefetch mechanism for accelerating general-purpose computation, 1994. 76

[49] P. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, 1967. 18

[50] J.L. Hennessy and D.A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann, 2007. 115

[51] Jay P. Hoeflinger and Bronis R. De Supinski. The openmp memory model. In *Proceedings of the First international workshop on OpenMP*, 2005. 25, 32

[52] H.P. Hofstee. Power Efficient Processor Architecture and The Cell Processor. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 258–262. IEEE Computer Society Washington, DC, USA, 2005. 5, 18, 134

[53] *Block-Matching in Motion Estimation Algorithms Using Streaming SIMD Extensions 3*. Intel Corporation, 2003. 102

[54] KL Johnson, MF Kaashoek, and DA Wallach. CRL: high-performance all-software distributed shared memory. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 213–226. ACM New York, NY, USA, 1995. 58

[55] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th annual international symposium on Computer Architecture*, pages 364–373. ACM, 1990. 76

[56] J.A. Kahle, M.N. Day, H.P. Hofstee, C.R. Johns, T.R. Maeurer, and D. Shippy. Introduction to the Cell Multiprocessor. *IBM Journal of Research and Development*, 49(4/5):589, 2005. 5, 18, 134

[57] M. Kandemir, O. Ozturk, and M. Karakoy. Dynamic on-chip memory management for chip multiprocessors. In *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 14–23, 2004. 5

[58] T. Kanungo, D.M. Mount, N.S. Netanyahu, C.D. Piatko, R. Silverman, and A.Y. Wu. An efficient k-means clustering algorithm: Analysis and implementation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 881–892, 2002. 18

[59] M. Kistler, M. Perrone, and F. Petrini. Cell multiprocessor communication network: Built for speed. *Micro, IEEE*, 26(3):10–23, May-June 2006. 18, 81, 86, 134

[60] S. Larsen, E. Witchel, and S. Amarasinghe. Increasing and Detecting Memory Address Congruence. In *Proceedings of the 11th International Conference on Parallel Architectures and Compilation Techniques*, pages 18–29, 2002. 102

[61] D.M. Lavery and W.W. Hwu. Modulo scheduling of loops in control-intensive non-numeric programs. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, volume 2, pages 126–137, 1996. 62

[62] Jacob Leverich, Hideho Arakida, Alex Solomatnikov, Amin Firoozshahian, Mark Horowitz, and Christos Kozyrakis. Comparing memory systems for chip multiprocessors. In *Proceedings of the 34th annual international symposium on Computer architecture*, pages 358–368, 2007. 5

[63] Mikko H. Lipasti, William J. Schmidt, Steven R. Kunkel, and Robert R. Roediger. Spaid: software prefetching in pointer- and call-intensive environments. In *Proceedings of the 28th annual international symposium on Microarchitecture*, pages 231–236. IEEE Computer Society Press, 1995. 76

[64] T. Liu et al. DBDB: Optimizing DMA Transfer for the Cell BE Architecture. In *Proceedings of the 23rd international conference on Supercomputing*, pages 36–45, 2009. 41

[65] Honghui Lu, Alan L. Cox, Sandhya Dwarkadas, Ramakrishnan Rajamony, and Willy Zwaenepoel. Compiler and software distributed shared memory support for irregular applications. In *Proceedings of the sixth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '97, pages 48–56. ACM, 1997. 59

[66] Chi-Keung Luk and Todd C. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 222–233. ACM, 1996. 76

[67] P.R. Luszczek, D.H. Bailey, J.J. Dongarra, J. Kepner, R.F. Lucas, R. Rabenseifner, and D. Takahashi. The hpc challenge (hpcc) benchmark suite. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 213. Citeseer, 2006. 10, 17

[68] Nir Magen, Avinoam Kolodny, Uri Weiser, and Nachum Shamir. Interconnect-power dissipation in a microprocessor. In *Proceedings of the 2004 international workshop on System level interconnect prediction*, pages 7–13, 2004. 5

[69] T.G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, et al. The 48-core scc processor: the programmer's view. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2010. 5

[70] J.D. McCalpin. Stream: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, 1995. URL `http://www.cs.virginia.edu/stream/`. 17

[71] J.E. Miller and A. Agarwal. Software-based instruction caching for embedded processors. In *Proceedings of the 2006 ASPLOS Conference*, volume 34, pages 293–302. ACM New York, NY, USA, 2006. 41

[72] V. Milutinovic et al. The Split Spatial/Non-Spatial Cache: A Performance and Complexity Evaluation. *Newsletter of Technical Committee on Computer Architecture*, pages 3–10, 1999. 126

[73] G. Moore. Excerpts from a conversation with gordon moore: Moores law. *Interview by Intel Corporation*. 3

[74] G. Moore et al. Cramming more components onto integrated circuits. *Electronics*, 38 (8), April 19 1965. 3

[75] C. Moritz, M. Frank, and S. Amarasinghe. Flexcache: A framework for flexible compiler generated data caching. In *Intelligent Memory Systems: Second International Workshop, IMS 2000, Cambridge, MA, USA, November 12, 2000: Revised Papers*. Springer, 2001. 42

[76] C.A. Moritz, M. Frank, W. Lee, and S. Amarasinghe. Hot Pages: Software Caching for Raw Microprocessors. *M1T-LCS Technical Memo LCS-TM*, 599, 1999. 42

[77] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, pages 62–73. ACM, 1992. 76

[78] Richard Murphy. On the effects of memory latency and bandwidth on supercomputer application performance. In *Proceedings of the 2007 IEEE 10th International Symposium on Workload Characterization*, pages 35–43, 2007. 5

[79] D. Nuzman and R. Henderson. Multi-platform Auto-vectorization. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 281–294, 2006. 80, 102

[80] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 2–11, 1996. 3

[81] Y. Paek, J. Hoeflinger, and D. Padua. Efficient and precise array access analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(1):65–109, 2002. 33, 52

[82] S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the 21st annual international symposium on Computer architecture*, pages 24–33. IEEE Computer Society Press, 1994. 76

[83] Guillermo Payá-Vayá, Javier Martín-Langerwerf, Sören Moch, and Peter Pirsch. An enhanced dma controller in simd processors for video applications. In *Proceedings of the 22nd International Conference on Architecture of Computing Systems*, pages 159–170, 2009. 103

[84] D. Pham, S. Asano, M. Bolliger, M.N. Day, H.P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The design and implementation of a first-generation cell processor. *IEEE International Solid-State Circuits Conference, Digest of Technical Papers*, pages 184–592 Vol. 1, Feb. 2005. 134

[85] D.C. Pham, T. Aipperspach, D. Boerstler, M. Bolliger, R. Chaudhry, D. Cox, P. Harvey, P.M. Harvey, H.P. Hofstee, C. Johns, et al. Overview of the architecture, circuit design, and physical implementation of a first-generation cell processor. *IEEE Journal of Solid-State Circuits*, 41(1):179–196, 2006. 134

[86] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. LU Decomposition and Its Applications. *Numerical Recipes in FORTRAN: The Art of Scientific Computing*, pages 34–42, 1992. 18

[87] B.R. Rau. Iterative modulo scheduling: an algorithm for software pipelining loops. In *Proceedings of the 27th annual international symposium on Microarchitecture*, pages 63–74. ACM New York, NY, USA, 1994. 62

[88] BR Rau, MS Schlansker, and PP Tirumalai. Code Generation Schema For Modulo Scheduled Loops. In *Microarchitecture, 1992. MICRO 25., Proceedings of the 25th Annual International Symposium on*, pages 158–169, 1992. 62

[89] R. Ravindran et al. Compiler-managed partitioned data caches for low power. In *Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 237–247, 2007. 126

[90] A. Rico, A. Duran, F. Cabarcas, Y. Etsion, A. Ramirez, and M. Valero. Trace-driven simulation of multithreaded applications. In *2011 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 87–96. IEEE, 2011. 21

[91] A. Rico, F. Cabarcas, C. Villavieja, M. Pavlovic, A. Vega, Y. Etsion, A. Ramirez, and M. Valero. On the simulation of large-scale architectures using multiple application abstraction levels. In *7th International Conference on High-Performance and Embedded Architectures and Compiler (HiPEAC)*, 2012. 21

[92] Alejandro Rico, Felipe Cabarcas, Antonio Quesada, Milan Pavlovic, Augusto Javier Vega, Carlos Villavieja, Yoav Etsion, and Alex Ramirez. Scalable simulation of decoupled accelerator architectures. Technical Report UPC-DAC-RR-2010-14, UPC, 2010. URL `http://gsi.ac.upc.edu/reports/2010/14/tasksim.pdf`. 20, 21

[93] J.A. Rivers et al. Reducing conflicts in direct-mapped caches with a temporality-based design. In *Parallel Processing, 1996., Proceedings of the 1996 International Conference on*, pages 154–163, 2002. 126

[94] A. Ros, M.E. Acacio, and J.M. Garcia. Cache Coherence Protocols for Many-Core CMPs. 5

[95] Amir Roth, Andreas Moshovos, and Gurindar S. Sohi. Dependence based prefetching for linked data structures. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 115–126. ACM, 1998. 76

[96] R. Rugina and M.C. Rinard. Pointer analysis for structured parallel programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(1):70–116, 2003. 52

[97] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, et al. Larrabee: a many-core x86 architecture for visual computing. In *Proceedings of the International Conference on Computer Graphics and Interactive Techniques*, pages 1–15, 2008. 103

[98] S. Seo et al. Design and implementation of software-managed caches for multicores with local memory. In *Proceedings of the 15th International Symposium on High Performance Computer Architecture*, pages 55–66, 2009. 41, 125

[99] A. Shahbahrami, B. Juurlink, and S. Vassiliadis. Performance Impact of Misaligned Accesses in SIMD Extensions. In *Proceedings of the 17th Annual Workshop on Circuits, Systems and Signal Processing*, pages 23–24, November 2006. 79, 82

[100] Z. Shen, Z. Li, and P.C. Yew. An empirical study of Fortran programs for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):356–364, 1990. 52

[101] J.L. Shin, Dawei Huang, B. Petrick, Changku Hwang, K.W. Tam, A. Smith, Ha Pham, Hongping Li, T. Johnson, F. Schumacher, A.S. Leon, and A. Strong. A 40 nm 16-core 128-thread sparc soc processor. *IEEE Journal of Solid-State Circuits*, 46(1):131–144, 2011. 4

[102] R.L. Sites. *Alpha Architecture Reference Manual*. Digital Press, 1998. 102

[103] Nathan Slingerland and Alan Jay Smith. Measuring the performance of multimedia instruction sets. *IEEE Transactions on Computers*, 51(11):1317–1332, 2002. 79

[104] S.B. Stepanian. SB-1 MIPS64 CPU Core. In *Embedded Processor Forum*, 2000. 80, 102

[105] Jimmy Su and Kathy Yelick. Array prefetching for irregular array accesses in titanium. In *In Sixth Annual Workshop on Java for Parallel and Distributed Computing*, 2004. 76

[106] D. Sweetman. *See MIPS Run*. Morgan Kaufmann, 2006. 102

[107] O. Takahashi, R. Cook, S. Cottier, SH Dhong, B. Flachs, K. Hirairi, A. Kawasumi, H. Murakami, H. Noro, H. Oh, et al. The circuits and physical design of the synergistic processor element of a cell processor. In *Symposium on VLSI Circuits. Digest of Technical Papers.*, pages 20–23. IEEE, 2005. 134

[108] J.M. Tendler, J.S. Dodson, JS Fields, H. Le, and B. Sinharoy. Power4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–25, 2002. 3

[109] *User Manual SPRU732C: TMS320 C64x/C64x+ DSP CPU and Instruction Set Reference Guide*. Texas Instruments, 2005. 80, 102

[110] S. Udayakumaran, A. Dominguez, and R. Barua. Dynamic allocation for scratch-pad memory using compile-time decisions. *ACM Transactions on Embedded Computing Systems (TECS)*, 5(2):472–511, 2006. 42

[111] Osman S. Unsal et al. Cool-cache for hot multimedia. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 274–283, 2001. 125

[112] J.W. VAN De Waerdt, S. Vassiliadis, S. Das, S. Mirolo, C. Yen, B. Zhong, C. Basto, J.P. van Itegem, D. Amirtharaj, K. Kalra, et al. The TM3270 Media-processor. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, page 12, 2005. 80, 102

[113] Steven P. Vanderwiel and David J. Lilja. Data prefetch mechanisms. *ACM Computing Surveys (CSUR)*, 32:174–199, June 2000. 76

[114] Hangsheng Wang, Li-Shiuan Peh, and Sharad Malik. Power-driven design of router microarchitectures in on-chip networks. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, 2003. 5

[115] M. Ware, K. Rajamani, M. Floyd, B. Brock, J.C. Rubio, F. Rawson, and J.B. Carter. Architecting for power management: The ibm® *power*$7^{TM}$ approach. In *IEEE 16th International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–11, 2010. 4

[116] J.D. Warnock, J.M. Keaty, J. Petrovick, J.G. Clabes, C.J. Kircher, B.L. Krauter, P.J. Restle, B.A. Zoric, and C.J. Anderson. The circuit and physical design of the power4 microprocessor. *IBM Journal of Research and Development*, 46(1):27–51, 2002. 3

[117] R.P. Wilson and M.S. Lam. Efficient context-sensitive pointer analysis for C programs. *ACM SIGPLAN Notices*, 30(6):1–12, 1995. 52

[118] E. Witchel, S. Larsen, C.S. Ananian, and K. Asanovic. Direct Addressed Caches for Reduced Power Consumption. In *International Symposium on Microarchitecture: Proceedings of the 34 th annual ACM/IEEE international symposium on Microarchitecture*, volume 1, pages 124–133, 2001. 42, 125

[119] P. Wu, A.E. Eichenberger, and A. Wang. Efficient SIMD Code Generation for Runtime Alignment and Length Conversion. In *Proceedings of the international symposium on Code generation and optimization*, pages 153–164, 2005. 81, 82, 102

[120] M. Yourst. Ptlsim users guide and reference, 2007. 21

[121] Matt T. Yourst. PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. In *Proceedings of the 7th International Symposium on Performance Analysis of Systems and Software*, pages 23–34, 2007. 21

[122] Zheng Zhang and Josep Torrellas. Speeding up irregular applications in shared-memory multiprocessors: Memory binding and group prefetching. In *In Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 188–200. ACM Press, 1995. 76