ARCHITECTING, PROGRAMMING, AND EVALUATING AN ON-CHIP INCOHERENT MULTI-PROCESSOR MEMORY HIERARCHY

BY

WOOIL KIM

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Doctoral Committee:

Professor Josep Torrellas, Chair and Director of Research
Professor Marc Snir
Professor David Padua
Professor William Gropp
Professor P. Sadayappan, Ohio State University

Abstract

New architectures for extreme-scale computing need to be designed for higher energy efficiency than current systems. The DOE-funded Traleika Glacier architecture is a recently-proposed extreme-scale manycore that radically simplifies the architecture, and proposes a cluster-based on-chip memory hierarchy without hardware cache coherence. Programming for such an environment, which can use *scratchpads* or *incoherent caches*, is challenging. Hence, this thesis focuses on architecting, programming, and evaluating an on-chip incoherent multiprocessor memory hierarchy.

This thesis starts by examining incoherent multiprocessor caches. It proposes ISA support for data movement in such an environment, and two relatively user-friendly programming approaches that use the ISA. The ISA support is largely based on writeback and self-invalidation instructions, while the programming approaches involve shared-memory programming either inside a cluster only, or across clusters. The thesis also includes compiler transformations for such an incoherent cache hierarchy.

Our simulation results show that, with our approach, the execution of applications on incoherent cache hierarchies can deliver reasonable performance. For execution within a cluster, the average execution time of our applications is only 2% higher than with hardware cache coherence. For execution across multiple clusters, our applications run on average 20% faster than a naive scheme that pushes all the data to the last-level shared cache. Compiler transformations for both regular and irregular applications are shown to deliver substantial performance increases.

This thesis then considers scratchpads. It takes the design in the Traleika Glacier architecture and performs a simulation-based evaluation. It shows how the hardware exploits available concurrency from parallel applications. However, it also shows the limitations of the current software stack, which lacks smart memory management and high-level hints for the scheduler.

To my love, EJ

Acknowledgments

My PhD journey was full of challenges. I was able to overcome obstacles with help and support from many people. I would like to express my sincere gratitude to all of them.

First of all, I would like to thank my academic advisor, Professor Josep Torrellas. He helped me not only academically but also mentally. Weekly or more frequent meeting with him drove me to the right direction. His advice led me to the completion of PhD journey. His endless effort, pursuit of perfection, and enthusiasm for research will be my goal throughout my life.

I would also like to thank my committee members. Professor Marc Snir provided very critical but insightful comments that lead my thesis to the more complete and accurate form. Professor William Gropp also gave comments about the finest detail. Professor David Padua was with all my exams from the qualifying exam to the final exam. Professor P. Sadayappan was a great collaborator and helper for many of my work. I cannot imagine my thesis without his help.

I also have to thank my collaborators in the projects that I was participated in. Sanket Tavarageri was a great collaborator. We shared lots of discussion and ideas, and his help was crucial in the completion of my work. Chunhua Liao was not only a nice collaborator but also a helpful mentor. I also want to mention Bala Seshasayee among X-Stack Treleika Glacier project members. His prompt answers and responses enabled progress in spite of many obstacles in the project.

My work would not be possible without help from IACOMA group members. I thank all IACOMA group members for their critical comments during the seminar and insightful suggestions. Besides, I thank Shanxiang, Yuelu, Prabhat, Tom, and Jiho for the time to chat about research, future, career, and culture. In fact, my PhD studies started from Wonsun's help. I appreciate his help and hope a great success in his continuing academic journey.

Discussion with other friends in the same field was always fun and helpful. Heeseok listened to all my crazy ideas and provided precious comments. Neal helped me for soft-landing to the university research.

Yun helped me to verify my idea. Liwen was also a good discussion partner for everything.

I met many friends here in Champaign. The time with them supported me. I thank Soobae, Hyunduk, Jaesik, Jintae, Choonghwan, Hyojin, Daehoon, Minje, Kyungmin, and Daejun. Julie's help for English and Professor Thomas Frazzetta's kindness was another support for my life in Champaign. I also thank Rigel group members for their help.

I also appreciate help and support from colleagues and bosses in Samsung Electronics. Nakhee and Jinpyo have been always my role models in all aspects of research and development. I really thank my previous bosses, Seh-Woong Jeong, Jaehong Park, and Steve Kwon. Without their help, I could not even start my PhD journey.

Lastly, I thank my family. I especially thank my parents. Their endless devotion made me stand here. I appreciate all their help and dedicate this dissertation to them. I appreciate support from my wife Eunjeong. She made my life in Champaign complete and successful.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Continuous progress in transistor integration keeps delivering manycore chips with ever-increasing transistor counts. For these chips to be cost-effective in a not-so-distant future — for example, as nodes of an exascale machine [35] — they will need to operate in a much more energy-efficient manner than they do today, following the ideas of what has been called Extreme-Scale Computing [30].

One such extreme-scale architecture is Runnemede [14], which is a processor architecture adopted in the Traleika Glacier (TG) project [31]. A Runnemede chip integrates about 1,000 cores and runs at low supply voltage. One of the keys to its expected high energy efficiency is a radically-simplified architecture. For example, it employs narrow-issue cores, hierarchically organizes them in clusters with memory, provides a single address space across the chip, and uses an on-chip memory hierarchy that does not implement hardware cache coherence. It includes incoherent caches and scratchpads.

An on-chip memory hierarchy without hardware coherence may have advantages in ease of chip implementation and, for some of the applications expected to run on these machines, even in energy efficiency. However, programming for it presents a non-trivial challenge. Indeed, the compiler or programmer has to orchestrate the movement of data between the different cache/memory levels. Moreover, although one can argue that large exascale machines such as a Traleika Glacier system will mostly run regular numerical codes, this is not the whole story. A single Runnemede chip, with so much compute power, is also an enticing platform on which to run smaller, more irregular programs. If so, the question remains as to how to program for such a memory hierarchy.

There are several other proposals of multiprocessor cache hierarchies without hardware coherence or with simplified hardware coherence. Two examples are Rigel [33] and Cohesion [34], which are manycores to accelerate regular visual applications. There is also substantial work on software cache coherence schemes, mostly relying on detailed compiler analysis of the code (e.g., [16, 18, 20]), and also using bloom filters to summarize communication [7]. Moreover, there are efforts that try to simplify the design of hard-

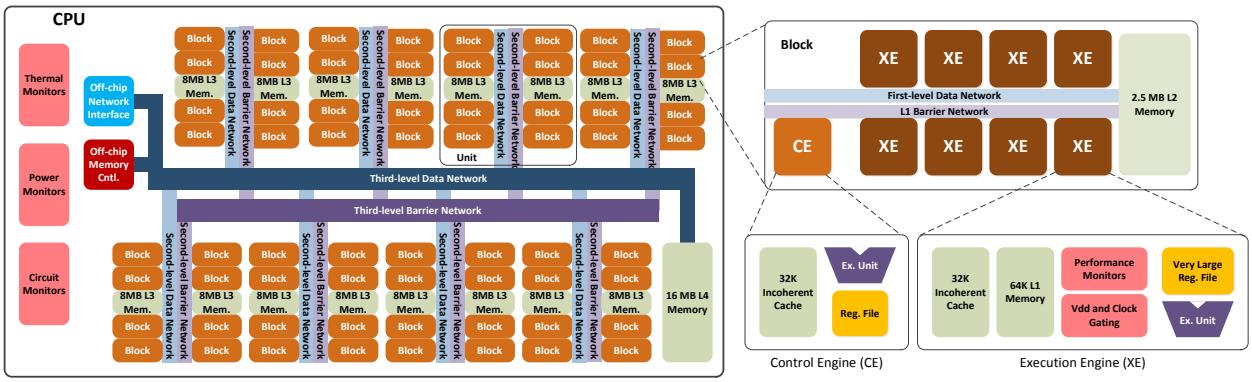

Figure 1.1: Runnemede chip architecture from [14].

ware cache-coherence protocols (e.g., [17, 45, 49]).

While we can learn from this work, the goal of this thesis is slightly different. One goal is to improve on how to use incoherent cache hierarchies. Hence, I design the Instruction Set Architecture (ISA), compiler, and a relatively user-friendly programming environment to use a cluster-based incoherent cache hierarchy like Runnemede's. Compared to past work, my goal is not seeking to simplify or minimize hardware cache coherence protocols. Instead, I am trying to exploit this novel cache hierarchy using, as much as possible, existing cache structures and programming styles.

A second goal of this thesis is to understand how well scratchpads work. The scratchpad structure of Runnemede is evaluated using its full software stack. The software stack is fairly elaborated and has been developed by assembling pieces from different parties. The evaluation is done in terms of performance and energy efficiency.

1.1 Motivation

Runnemede [14] is an extreme-scale manycore recently proposed by Intel for the 2018-2020 timeframe. The chip integrates about 1,000 cores, and its goal is to use energy very efficiently. To do so, the manycore operates at low supply voltage, and can change the voltage and frequency of groups of cores separately.

In addition, one of the keys to higher energy efficiency is a drastically-simplified architecture. Specifically, cores have a narrow-issue width and are hierarchically grouped in clusters with memory. Functional core heterogeneity is avoided and, instead, the chip relies on voltage and frequency changes to provide heterogeneity. All cores share memory and a single address space, but the on-chip memory hierarchy has no

support for hardware cache coherence. Figure 1.1 shows the manycore's architecture from [14].

As shown in the figure, each core (called XE) has a private cache, which is called L1. This is a normal cache but without hardware-coherence support. A core also has a scratchpad labeled in the figure as L1 Memory. A group of 8 cores forms a cluster called a *Block*, which has a second level of SRAM called L2. Multiple blocks are grouped into a second-level cluster called *Unit*, which also has a third level of SRAM called L3. The system can build up hierarchically further, but I will disregard it in this thesis for simplicity of explanation.

In Runnemede, L2 and L3 are scratchpad memories rather than caches, each with their own range of addresses. In this thesis, I will first assume they are incoherent caches (Chapter 2) and then that they are scratchpads (Chapter 3).

The fact that the cache hierarchy is not coherent means that the application (i.e., the programmer or compiler) has to orchestrate the data movement between caches. This may be seen as a way to minimize energy consumption by eliminating unnecessary data transfers — at least for the more regular types of applications that are expected to run on this machine. However, it introduces a programming challenge.

In the rest of this thesis, I first focus on a hierarchy with incoherent caches. I propose an ISA support, compiler, and programming model for a cluster-based incoherent cache hierarchy like the one described. Then, I focus on a memory hierarchy with scratchpads only. I evaluate the scratchpad hierarchy with all its software stack.

Chapter 2

Architecting, Programming, and Evaluating Incoherent Caches

In this chapter, we seek to understand how to program an incoherent multi-processor cache hierarchy, and what hardware support is required to get a balance between performance and programmability.

The contributions of this chapter are as follows:

• We propose a simple ISA support to manage an incoherent cache hierarchy, largely based on *writeback* and *self-invalidation* instructions. We show the subtle issues that these instructions need to face, including reordering in the pipeline, and effective use of caches organized in clusters for energy efficiency.

• We show two relatively user-friendly programming approaches that use this ISA to program the machine. The programming approaches involve shared-memory programming either inside a cluster only, or across clusters. They rely on annotating synchronization operations, and on identifying producer-consumer pairs.

• Our simulation results show that, with our approach, the execution of applications on incoherent cache hierarchies can deliver reasonable performance. For execution within a cluster, the average execution time of our applications is only 2% higher than with hardware cache coherence. For execution across multiple clusters, our applications run on average 20% faster than a naive scheme that pushes all the data to the last-level shared cache.

This chapter is organized as follows. Section 2.1 describes the basic ISA architecture proposed; Sections 2.2 and 2.3 present our two programming approaches; Sections 2.5 and 2.6 evaluate our design.

2.1 Basic Architectural Support

2.1.1 Using Incoherent Caches

In the architecture considered, all cores share memory but the cache hierarchy is not hardware-coherent. Writes by a core are not automatically propagated to other cores because caches do not snoop for coherence requests, nor is there any directory entity that maintains the sharing status of data. For a producer and a

consumer core to communicate the value of a variable, we need two operations. First, after the write, the producer needs to export the value to a shared cache that the other core can see. Then, before the read, the consumer needs to prepare for importing the fresh value from the shared cache.

The first operation is done by a *writeback* (*WB*) operation that copies the variable's value from the local cache to the shared cache. The second operation is a *self-invalidation* (*INV*) that eliminates a potentially stale copy of the variable from the local cache. Between the WB and the INV, the processors need to synchronize. Figure 2.1 shows the sequence.

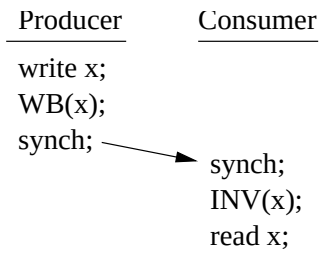

Figure 2.1: Communication between incoherent caches.

We call the set of dynamic instructions between two consecutive synchronizations of a thread an *epoch*. For correct operation, at the beginning of an epoch, a thread self-invalidates any data that it may read in the epoch and that may be stale because of an earlier update by other threads. At the end of the epoch, the thread writes-back any data that it may have written in the epoch and that may be needed by other threads in later epochs.

Except for the lack of automatic coherence, cache hierarchies operate as usual. They use lines to benefit from spatial locality but they do not need to support inclusivity.

2.1.2 Instructions for Coherence Management

WB and INV are memory instructions that give commands to the cache controller. For this initial discussion, assume that each processor has a private L1 cache and there is a shared L2 cache. WB and INV can have different flavors. First, they can operate on different data granularities: byte, word, double word, or quad word. In this case, they take as an argument the address of the byte, word, double word, or quad word. Second, WB and INV can also operate on a range of addresses. In this case, they take two registers, one with the start address and the other with the length. Finally, with the mnemonic WB ALL and INV ALL, they operate on the whole cache. In this case, there is no argument.

Since caches are organized into lines for spatial locality, WB and INV internally operate at cache line granularity. Specifically, WB writes back all the cache lines that overlap with the address or address range specified. Consequently, when two different cores write to different variables in the same memory line and then each performs a WB on its variable, we need to prevent the cores from overwriting each other's result. To prevent it, and to minimize data transfer volume, we use fine-grained dirty bits in the cache lines. If our finest grain is bytes, then we use per-byte Dirty (D) bits. When a WB is executed, the cache controller reads the target line(s) and only writes back the dirty bytes. WB has no effect if the target addresses contain no modified valid data. After a line is written back, it is left in unmodified valid state in the private cache.

INV eliminates from the cache all the cache lines that overlap with the address or address range specified. Since a cache line has a single Valid (V) bit, all the words in the line need to be eliminated. If the line contains some dirty bytes, the cache controller first writes them back to the shared cache before invalidating the line. If there are no valid cached lines in the target addresses, INV has no effect.

Overall, WB and INV do not cause the loss of any updated data that happens to be co-located in a target cache line. Hence, the programmer can safely use WB and INV with variables or ranges of variables, which the hardware expands to cache line boundaries. The lines can include non-shared data. At worst, we trigger unnecessary moves.

2.1.3 Instruction Reordering

The INV and WB instructions introduce some reordering constraints with respect to loads and stores to *the same address*. The important constraints are INV versus load, and WB versus store. The minimum set of ordering constraints is INV $\rightarrow$ load and store $\rightarrow$ WB. Reordering of these two sequences results in reading stale value or not exposing modified value. Thus, they must not be reordered in any case. We consider more ordering constraints to respect programmers' intention.

Consider INV first. Neither loads that precede an INV nor loads that follow the INV can be reordered by the compiler or hardware relative to the INV (Figure 2.2a). Effectively, a thread uses *INV(x)* to refresh its view of x and, therefore, reordering would imply that a load sees a different view than the one the program intended.

Consider now a WB. Neither stores that precede the WB nor stores that follow the WB can be reordered by the compiler or hardware relative to the WB (Figure 2.2b). The reason is that a *WB(x)* posts the value of

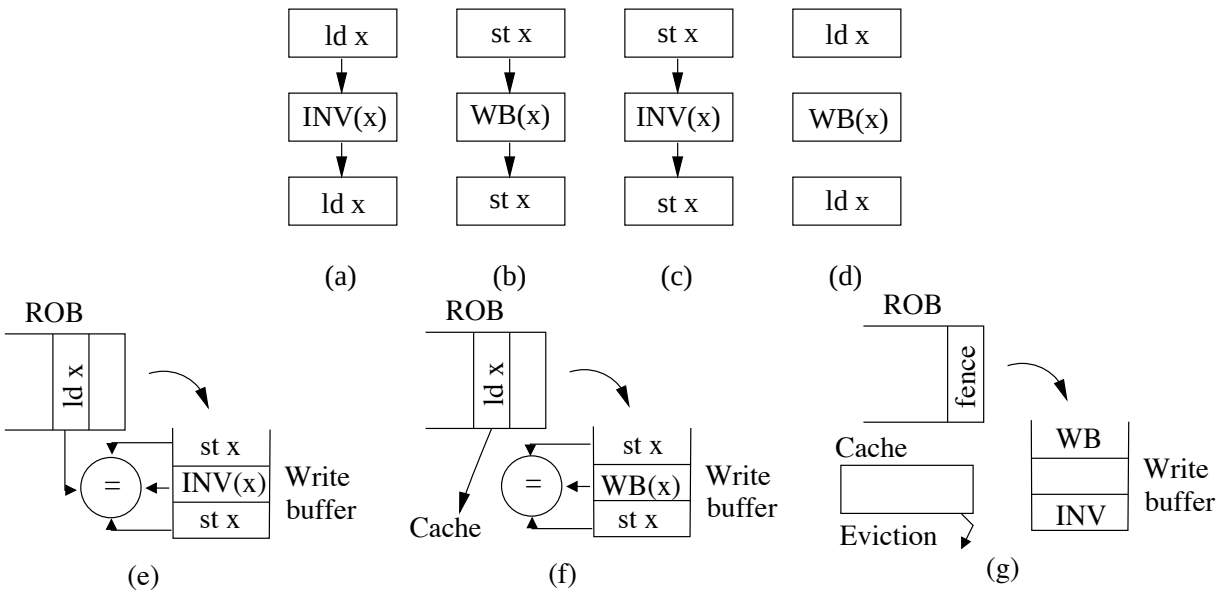

Figure 2.2: Ordering constraints.

x globally and, therefore, reordering would imply that the value posted would be different than the one the program intended.

Reordering stores that precede or follow INV is also disallowed. An INV always writes-back dirty data before invalidating the line. Consequently, no data is lost. However, it also means that a reorder can affect the time when a value is posted globally, and the value posted, as in the WB case. Consequently, we disallow such reordering as well (Figure 2.2c).

Finally, loads that precede or follow a WB can always be reordered (Figure 2.2d). This is because a WB does not change the value of the line in the local cache. Hence, a reodered load sees the same value. Note that this ability to pick a load that follows a WB to the same address and execute it before the WB can help performance.

These reordering requirements can be easily enforced in hardware. INV moves in the pipeline like a store. Consequently, by the time an *INV(x)* in the write buffer initiates its cache update, prior loads to *x* will be retired and prior stores to *x* will be completed. Moreover, no subsequent load or store to *x* will be allowed to initiate its cache access for as long as *INV(x)* is in the write buffer (Figure 2.2e).

WB is also placed in the write buffer like a write. Stores to *x* before and after *WB(x)* will follow program order. Loads to *x* that follow *WB(x)* will be allowed to proceed to the cache earlier (Figure 2.2f).

Finally, we consider the point when a WB or an INV instruction is complete. This point is significant

because only then can a fence instruction following the WB or INV instruction complete (Figure 2.2g). On the surface, an INV completes when the L1 cache acknowledges that the line has been invalidated, and a WB when the L1 confirms that the line has been safely written back to L2. In reality, however, we need to be more conservative and wait for any current cache eviction to be completed before claiming that the WB or INV has completed. The reason is that, if the line that the WB or INV is supposed to write back or invalidate, respectively, was dirty and is currently being evicted to L2, the WB or INV instruction will not find it in L1, and we will claim WB or INV completion. However, the intended operation is not completed until the L2 is updated. Consequently, at any time, we keep a counter of the lines that are currently being written back. A fence after a WB or INV will only complete when the WB or INV are complete and the eviction counter is zero.

2.1.4 Synchronization

Since conventional implementations of synchronization primitives rely on the cache coherence protocol, they cannot be used in machines with incoherent cache hierarchies. Any lock release would have to be followed by a WB, and spinning for a lock acquire would require continuous cache misses because each read would be preceded by INV.

To avoid such spinning over the network, machines without hardware cache coherence such as the Tera MTA, IBM RP3, or Cedar, have provided special hardware synchronization in the memory subsystem. Such hardware often queues the synchronization requests coming from the cores, and responds to the requester only when the requester is finally the owner of the lock, the barrier is complete, or the flag condition is set. All synchronization requests are uncacheable. The actual support in Runnemede [14] is not published, but it may follow these lines.

In this paper, we place the synchronization hardware in the controller of the shared caches. We provide three synchronization primitives: barriers, locks, and conditions. When a synchronization variable is declared, the L2 cache controller allocates an entry in a synchronization table and some storage in the controller's local memory. When a processor issues a barrier request, the controller intercepts it, and only responds when the barrier is complete. Similarly, for a lock acquire request or a condition flag check, the controller intercepts the requests and only responds when it is the requester's turn to have the lock, or when the condition is true. In addition, the controller releases the synchronization when timeout occurs to avoid a

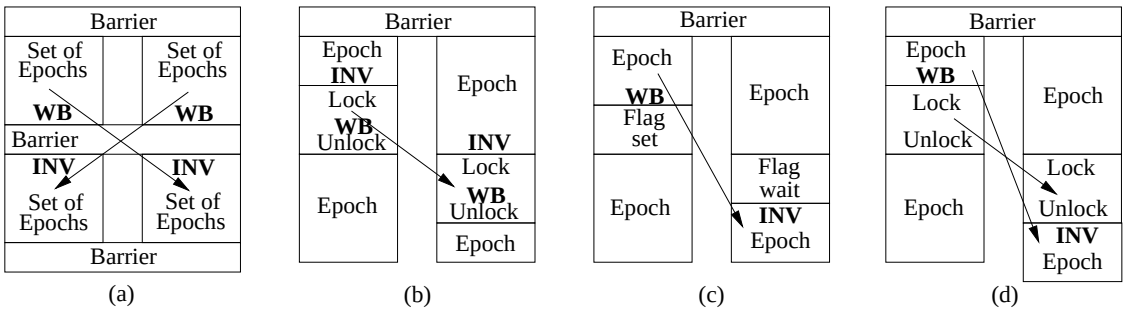

Figure 2.3: Annotations for communication patterns enabled by: barriers (a), critical sections (b), flags (c), and dynamic happens-before epoch orderings (d).

hardware deadlock.

Since synchronization support is not the focus of this paper, we do not consider further details.

2.2 Programming Model 1: MPI + Shared Intra Block

The first programming model that we propose for this machine is to use a shared-memory model inside each block and MPI across blocks. In this case, the *MPI_Send* and *MPI_Recv* calls can be implemented cheaply. A sender and a receiver communicate message information using hardware synchronization through the last level cache. This eliminates receiver's busy-waiting with spinning over on-chip network and enables a rendezvous protocol between the sender and the receiver. Data transfer is done through common shared level cache, i.e. L3 cache. Specifically, an *MPI_Send* of variable *x* is translated into a copy of *x* to a location *x_buff* in an on-chip L2-uncacheable shared buffer. An *MPI_Recv* of variable *y* is translated into a copy from location *y_buff* in the L2-uncacheable shared buffer to location *y*. Moreover, in communication with multiple recipients such as a broadcast, there is no need to make multiple copies; a single copy to *x_buff* suffices. The multiple receivers will all read from the same location. Finally, we can implement non-blocking *MPI_Isend* and *MPI_Irecv* calls by using another thread in the core to perform the copies. One can implement a protocol in a similar way to [26].

In the rest of this section, we focus on intra-block shared-memory programming. We first describe the proposed approach and then some advanced hardware support.

2.2.1 Intra-block Programming

Ideally, we would like to use the compiler to analyze the program and automatically augment it with WB and INV instructions. For many programs, however, it is hard to get efficient compiler-generated instrumentation due to intensive use of pointers and dynamic communication patterns. For this reason, we develop a simple approach that is easily automatable, even if it does not attain optimal performance. Also, the programmer can use his knowledge of the program to improve its performance.

The approach is based on relying on the synchronization operations in the data-race-free (DRF) program. In the DRF program, the synchronization can be used as explicit markers of changes in communication patterns. Hence, at the point immediately before and after a synchronization operation, depending on the type of synchronization, WB and INV instructions are inserted. Our algorithm decides which instructions to add, and the programmer can refine or overwrite them. For the programs with data races, we consider them separately. While the behavior of C or C++ programs with data races is undefined, it is desirable to have a way to support them as programmers originally intended. We provide fine-grained WB/INV for handling such a case.

Recall that a block has a per-core private L1 cache and a shared L2 cache (Figure 2.4a). In addition, there is a one-to-one thread-to-core mapping and no thread migration for simplicity of discussion.

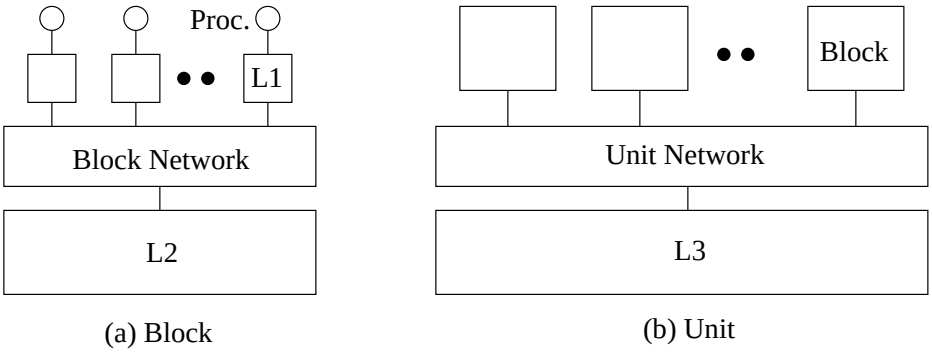

Figure 2.4: Single block (a) and multi-block (b) cache hierarchy.

Annotating Different Communication Patterns.

Each type of synchronization operation gives us a clue of the type of potential communication pattern that it manages. For instance, a program-wide barrier synchronization marks a point where a thread's post-barrier accesses can communicate with pre-barrier accesses from any of the other threads (Figure 2.3a).

Hence, the simplest scheme is as follows: immediately before the barrier, we place a WB for all the shared variables written since the last global barrier; immediately after the barrier, we place an INV for all the shared variables that will receive exposed reads (i.e., a read before any write) until the next global barrier.

In some cases, a thread owns part of the shared space, and reuses it across the barriers as if it was its private data. In this case, we do not need to write back and invalidate that part of the shared space. Also, the programmer can often provide information to reduce WB and INV operations. Finally, if the code between global barriers is long and we lack information on the sharing patterns, we use WB ALL and INV ALL, which write back and invalidate the entire L1 cache.

Threads often communicate by accessing variables inside a critical section protected by lock acquire and lock release (Figure 2.3b). In this case, immediately before the acquire, we place an INV for all the shared variables that will receive exposed reads inside the critical section. Moreover, immediately before the release, we place a WB for all the shared variables written inside the critical section. This basically conforms release consistency memory model. Note that we place the INV *before* the acquire rather than *after* to reduce the duration of the critical section.

Threads also coordinate with flag set-wait synchronization (Figure 2.3c). In this case, immediately before the signal, the producer thread needs a WB for all the shared variables written in the epochs since a global barrier or an equivalent point of full WB. Also, immediately after the wait, the consumer thread needs an INV for all the shared variables that will receive exposed reads in the epochs until the next global barrier or equivalent point of full INV. Like in the case of barriers, if such epochs are very long, we use WB ALL and INV ALL, respectively.

For other types of structured synchronization patterns that we have not seen in our applications, we can follow a similar methodology. However, there is also a general pattern of communication through dynamically-determined happens-before order — especially in irregular applications. After a thread completes a critical section, it may want to consume data that was generated by earlier holders of the critical section before they executed the critical section (Figure 2.3d). For example, this is observed with task-queue operations. A thread places a task in the task-queue and then another thread fetches the task and processes it. The task-queue is updated using a critical section, but there is significant communication outside the critical section as the consumer processes the task.

We call this pattern Outside Critical-Section Communication (OCC). Unless the programmer explicitly

states that there is no OCC, our programming model has to assume that there is. Hence, before a lock acquire, we add a WB of the shared variables written since a global barrier or point of full WB. After a lock release, we add an INV of all the exposed reads until the next global barrier or point of full INV. Often, these become WB ALL and INV ALL.

Discussion.

Many programs running on a Runnemede-style chip are likely to have a structure based on global barriers. In this case, it is easy for the compiler or programmer to insert WB and INV instructions.

Relying on explicit synchronization to orchestrate data transfers is insufficient when the program has data races. Indeed, the communication that data races induce in cache-coherent hierarchies is invisible in an incoherent cache hierarchy. Consider an example in Figure 2.5 with declaration of volatile variables and proper fence for memory models. Two processors try to communicate with a plain write and a spinloop without synchronization, but the spinloop may last forever (Figure 2.5a). This is because the consumer may not see the update.

```
// producer     // consumer
 data = 1;      for (; ; ) {
 flag = 1;        if (flag)
                    process (data);
                  }
```

(a)

```
// producer        // consumer
 data = 1;          for (; ; ) {
 WB (data);           INV (flag);
 fence;               if (flag) {
 flag = 1;              fence;
 WB (flag);             INV (data);
                        process (data);
                      }
                    }
```

(b)

Figure 2.5: Enforcing data-race communication.

If we want to support the data-race communication, we need to identify the data race and augment it with WB, INV, and fence instructions. Specifically, as shown in Figure 2.5b, the producer needs to follow each update with a WB, while the consumer needs to precede each read with an INV. Moreover, to prevent access reordering, we need a fence between the two WBs in the producer and the two INVs in the consumer.

The data-race communication pattern can be re-written by the programmer to use a hardware synchronization flag. In this case, the busy-waiting loop is changed to a hardware flag wait, while the producer write is changed to a hardware flag set.

Table 2.1: Classification of the benchmarks based on the communication patterns present.

Benchmark	Class	Barrier	Critical	Outside Critical	Data Race
FFT	Barrier	x			
LU	Barrier	x			
Cholesky	Outside Critical	x	x	x	x
Barnes	Barrier + Outside Critical	x	x	x	
Raytrace	Critical	x	x		x
Volrend	Barrier + Outside Critical	x		x	x
Ocean	Barrier + Critical	x	x		
Water	Barrier + Critical	x	x		

Application Classification.

Table 2.1 lists the applications that we use for intra-block programming in Section 2.6, and classifies them according to the communication patterns present. For each application, we list the main pattern under the *Main* column, and then other patterns that the application exhibits under *Other*. The patterns are those just described: barrier, critical section, flag, outside critical section, and data race.

From the table, we see that different applications have different dominant communication patterns. FFT and LU have mostly communication patterns enabled by barriers; Raytrace have patterns enabled by critical sections; Cholesky has the outside critical-section communication pattern. In addition, some applications have multiple patterns. Raytrace has asymmetric data races in its work pool structure. While a thread updates the work pool in a critical section, another thread reads it outside of any critical section. Overall, the data shows that our programming model needs to support at least all the communication patterns that we described earlier.

2.2.2 Advanced Hardware Support

Surrounding epochs with full cache invalidation (INV ALL) and writeback (WB ALL) results in unnecessarily low performance — especially for short epochs such as critical sections. One alternative is to explicitly list the variables or address ranges that need to be invalidated or written back. However, this approach is undesirable because it requires programming effort and programmers are prone to make mistakes. To improve both performance and programmability, we proposed two small hardware buffers called *Entry Buffers*. They can substantially reduce the cost of WB and INV in relatively-short epochs.

Modified Entry Buffer (MEB) for WB.

The MEB is a small hardware buffer that automatically accumulates the IDs of the cache lines that are being written in the epoch. Therefore, at the end of the epoch, when we need to write back the lines written in the epoch, we can use the MEB information. With this support, we avoid a traversal of the cache tags and the costly writeback of all the dirty lines currently in the cache. In short epochs, the MEB can save substantial overhead.

The MEB is designed to be cheap. It is small (e.g., 16 entries) and each entry only includes the ID of a line, rather than a line address. For example, for a 32-Kbyte cache with 64-byte lines, the ID is 9 bits. The MEB is updated in parallel with a write to the L1 cache. Specifically, every time that a clean byte is updated (recall that the cache has per-byte D bits), if the line's ID is not in the MEB, it is inserted.

Some entries in the MEB may become stale. This occurs when an MEB entry is created for a line that is written to, and later the line is evicted from the cache by another line that is never written to. To simplify the MEB design, stale entries are not removed. At the end of the epoch, as the MEB is traversed, only dirty lines are written back.

We use the MEB in small critical sections that conclude with WB ALL because the programmer did not provide additional information. The MEB records all the lines written in the critical section. When we reach WB ALL at the end of the epoch, the MEB is used, potentially saving many writebacks. However, if the MEB overflows during critical section execution, the WB ALL executes normally.

Invalidated Entry Buffer (IEB) for INV.

At the beginning of an epoch, we need to invalidate all the lines that the epoch will expose-read and that are currently stale in the local cache. Since explicitly listing such lines may be difficult, programmers may use INV ALL. This operation is very wasteful, especially in small epochs. Hence, we propose not to invalidate any addresses on entry, and to use the IEB instead.

The IEB is a small hardware buffer that automatically collects the addresses of memory lines that *do not* need to be invalidated on a future read. They do not need to because they have already been read earlier in the epoch and, at that point, they have been invalidated if they were in the cache. Hence, they are not stale. With the IEB, we can keep the number invalidations to a very small number.

The IEB only has a few entries (e.g., 4) because it is often searched and needs to be fast. Each entry has

the actual address of a line that needs no invalidation on a future read. The IEB contains exact information, and is updated as follows. The IEB is accessed at every L1 read. If the line's address is already in the IEB, no action is taken. Otherwise, the action depends on the outcome of the cache access. Specifically, if the read hits in the cache and the target byte is dirty, no action is taken. This is because the byte was written in the past by the current core and, therefore, it is not stale. In all other cases, the line's address is added to the IEB. Moreover, if the read hits in the cache, the line is invalidated. Finally, in either hit or miss, the read gets a fresh copy of the line from the shared cache.

Unlike the MEB, the IEB does not store unneeded entries. Every read is checked against the IEB, to avoid invalidating the same line twice. Invalidations are expensive because they are followed by a cache miss in the critical path.

Finally, the IEB is most useful in short epochs like small critical sections. The reason is that if the IEB needs to track many lines, it may overflow. Each time that an IEB entry is evicted, if the corresponding line is accessed again, the hardware causes one unnecessary invalidation. The execution is still correct, but the performance decreases.

2.3 Programming Model 2: Shared Inter Block

The second programming model that we propose for this machine is to use a shared-memory model across all processors — irrespective of whether they are in the same block or in different ones (Figure 2.4b). Our main idea is that, to obtain performance, we need to have *level-adaptive* WB and INV instructions. This means that, if two threads that communicate end-up being mapped into the same block, then the WB and INV operate as described until now: WB writes the line back to L2 and INV invalidates the line from L1. However, if the two threads end-up being mapped into different blocks, then WB writes back the line all the way to L3, and INV invalidates the line from both L1 and L2. We require that an application annotated with level-adaptive WB and INV runs correctly both within a block and across blocks without any modification.

For completeness, we also provide an instruction that explicitly writes back lines to the L3 cache (and to the L2 in the process) and one that explicitly invalidates lines from the L2 cache (and from the L1 in the process). They are *WB_L3(Addr)* and *INV_L2(Addr)*, respectively.

In this section, we first describe the proposed programming approach and then the hardware support.

2.3.1 Inter-block Programming

In the programming approach of Section 2.2, relatively little information was needed to insert WB and INV instructions. In the approach presented here, we need two types of information to insert level-adaptive WB and INV. First, we need to know how the program's computation is partitioned and mapped into threads. We assume static mapping of computation to threads. For example, given a parallel loop, the relevant information may be that the iterations are logically grouped into N consecutive chunks (where N is the number of threads), and chunk i is executed by thread i. Note that we do not know how the threads themselves will map to physical processors at runtime (i.e., which processors and in what clusters). However, such mapping will not be allowed to change dynamically.

The second piece of information needed is the producer-consumer patterns. Specifically, we need to identify: (i) all the data that is produced by thread i and consumed by thread j, and (ii) what epochs in i and j produce and consume the data, respectively.

Once we know this information, we can annotate the code with the level-adaptive WB and INV instructions *WB_CONS (Addr, ConsID)* and *INV_PROD (Addr, ProdID)*. In these instructions, we augment the WB mnemonic with *CONS* for consumer, and give it as a second argument the ID of the consumer thread *ConsID*. Similarly, we augment the INV mnemonic with *PROD* for producer, and give it the ID of the producer thread *ProdID*.

For the example of threads i and j producing and consuming x, the instructions inserted are as follows (Figure 2.6). At the end of the epoch that produces x in thread i, we place *WB_CONS (x, j)*, while at the beginning of the epoch that consumes x in thread j, we place *INV_PROD (x, i)*. The implementation of these instructions is described in Section 2.3.2.

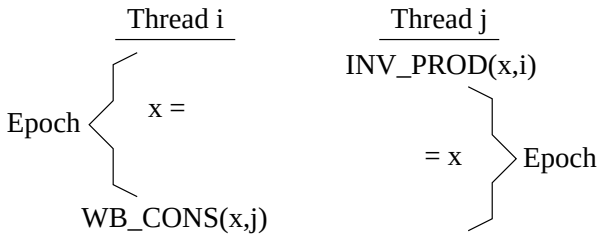

Figure 2.6: Example of level-adaptive WB and INV.

Generating this information requires deeper code analysis than in the first programming model (Section 2.2). Hence, while the first programming model can handle applications with pointers and irregular

structures, this second model is targeted to more compiler-analyzable codes. In our experiments of Section 2.6, we use OpenMP applications.

To be able to insert level-adaptive WB and INV instructions, the compiler starts by performing inter-procedural control flow analysis to generate an interprocedural control flow graph of the program. Then, knowing how the computation is partitioned into threads, the compiler performs data flow analysis to find pairs of producer and consumer epochs in different threads (i, j). Then, the compiler inserts *WB_CONS* in i and *INV_PROD* in j.

In the following, we present some details of our compiler algorithm for level-adaptive WB and INV. First, we present the general case; then we consider irregular applications.

Extracting Producer-Consumer Pairs.

Programs that have no pointer manipulation or aliasing, use OpenMP work-sharing constructs, and schedule OpenMP *for* loops statically are typically amenable to compiler analysis. In these programs, there may be data dependence between serial and parallel sections and between parallel sections. Typically, the relation between serial and parallel sections is relatively easy to analyze, while the relation between parallel sections is harder to analyze. It requires understanding the program's inter-procedural control flow and performing dataflow analysis.

Inter-procedural control flow analysis finds parallel *for* loop iterations that potentially communicate with each other. From each starting *for* loop, we traverse the control flow graph to find reachable *for* loops. Those *for* loops that are unreachable are not considered further. The reachable loops are now targets of dataflow analysis. We apply DEF-USE analysis from a preceding *for* loop as a producer to any reachable *for* loop as a consumer. We compare the array structures accessed by the producer and consumer loops to determine if any is in both loops. If an array is in both, we compare the indices. Since we use static scheduling, we know the mapping of iteration to thread ID. Consequently, the compiler can determine if there is a data dependence between two threads. In this case, it puts WB_CONS (address,consumerID) in the producer and INV_PROD (address, producerID) in the consumer.

We perform this analysis in a semi-automatic way using some extensions to the ROSE compiler infrastructure [43] that we developed. The analyzer gives two types of information: what OpenMP code section has producer-consumer relation, and whether this requires WB_CONS/INV_PROD. Programmers

insert WB/INV at proper places according to the information. Note that the consumer and producer IDs in the WB_CONS and INV_PROD instrumentation are typically expressed as equations of local thread ID.

Our approach is similar to interprocedural analysis used in auto parallelizing compiler [11, 28]. Our approach is also similar to the translation from OpenMP to MPI [8, 9] and to the implementation of software DSM [5]. They all use control flow and data flow analysis to find communicating data and producer-consumer pairs. In particular, inferring *MPI_Send* corresponds to inserting WB_CONS, and inferring *MPI_Recv* corresponds to inserting INV_PROD.

However, our approach differs from previous techniques in four aspects. First, our approach does not maintain explicit replicated copies of data, while translated MPI code or software DSM does. Second, our approach executes the serial section in only one thread, and the result is written back by WB to the global cache; translated MPI code executes the serial section in all nodes. Third, our approach implements single producer-multiple consumers with a single WB in the producer; MPI requires multiple *MPI_Send* if point-to-point communication is used. Finally, our approach is reasonably efficient even if exact dataflow analysis is not possible — e.g., when we do not know the exact consumers accurately. In this case, the producer writes back the data to the last level cache. In an MPI translation, we would need to send messages to all the possible receivers.

Handling Irregular Cases.

Many scientific applications use sparse computations that both are iterative and have data access patterns that remain the same across iterations. An example is the conjugate gradient method for solving systems of linear equations using sparse matrix-vector multiplications. For such codes, we use an inspector [21, 53] to gather information on irregular data accesses so that WB and INV instructions can be performed only where necessary. The inspector is inserted in the code and is executed in parallel by the threads. The cost of the inspector is amortized by the ensuing selective WB and INV.

Figure 2.7 shows an excerpt from conjugate gradient. The original code contained a loop that reads array *p[]* with indirect accesses (Line 19), and another loop that writes *p[]* (Line 27). We assume static scheduling of OpenMP loops with chunk distribution. Thus, continuous iterations are allocated for each thread.

The inspector loop is placed at the beginning (Line 9). It determines the ID of the writer thread that will

```
1  #pragma omp parallel
2  {
3    // inspector code starts
4    int total_threads=omp_get_num_threads();
5    int my_id=omp_get_thread_num();
6    int my_j_start=(n/total_threads)*my_id;
7    int my_j_end=(n/total_threads)*(my_id+1);
8    int target_id;
9    int i, j, k;  // all private by default policy of OpenMP
10
11   for (j=my_j_start; j<my_j_end; j++) {
12     for (k=A[j]; k<A[j+1]; k++) {
13       target_id = colidx[k] / (n/total_threads);
14       conflict[k] = target_id;
15     }
16   }
17   // inspector code ends
18
19   for (i=0; i<imax; i++) {
20     #pragma omp for
21     for (j=0; j<n; j++) {
22       for (k=A[j]; k<A[j+1]; k++) {
23         if (conflict[k] != my_id)
24           INV_PROD(&(p[colidx[k]]),conflict[k]);
25         read p[colidx[k]]; // indirect access
26     }
27     ...
28     #pragma omp for
29     for (j=0; j<n; j++) {
30       write p[j];
31     }
32     WB_L3_range(&(p[my_j_start]),
33                 my_j_end-my_j_start);
34     #pragma omp barrier
35   }
36 }
```

Figure 2.7: An iterative loop with irregular data references with the inspector.

produce the value obtained by each read. The result is stored in array *conflict*. Then, at every iteration of the loop that reads, it checks the ID of the writer (Line 21). If it is not the same as the reader's ID (Line 21), it inserts an INV_PROD instruction with the ID of the thread that will produce it (Line 22). Otherwise, it skips the INV.

In the figure, after the loop that writes to *p[]*, we place a WB to the L3 cache for the whole range written. We could perform a similar analysis like the one described for the read, and save some writes to L3 by using WB_CONS. For simplicity, we write everything to L3.

2.3.2 Hardware Support

We implement WB_CONS and INV_PROD as follows. In each block, the L2 cache controller has a *ThreadMap* table with the list of the IDs of the threads that have been mapped to run on this block. This table is filled by the runtime system when the threads are spawned and assigned to processors, typically at the beginning of the program.

When a thread executes the *WB_CONS(addr,ConsID)* instruction, the hardware checks the *ThreadMap* in the local L2 controller and determines whether or not the thread with ID *ConsID* is running in the same block. If it is, for each of the lines in *addr*, the hardware writes back the dirty bytes only to L2. Otherwise, the dirty bytes are written back to both L2 and L3. Note that, for a given line, this operation may require checking both the L1 and L2 tags.

When a thread executes the *INV_PROD(addr,ProdID)* instruction, the hardware checks the *ThreadMap* in the local L2 controller and determines whether or not the thread with ID *ProdID* is running in the same block. If it is, for each of the lines in *addr*, the hardware invalidates the line only from L1. Otherwise, the line is invalidated from both L1 and L2. For each line, this operation requires checking both the L1 and L2 tags.

Note that when a thread's WB_CONS propagates the target updates to L3, the other L1 caches in the same block, and the L1 and L2 caches in other blocks, may retain stale copies of the line. Similarly, when a thread's INV_PROD self-invalidates the target addresses from the L1 and L2 caches, the other L1s in the same block, and the L1 and L2 caches in other blocks, may keep stale values.

It is sometimes necessary for an epoch to perform a full WB or INV of the whole cache. Hence, we also support WB_CONS_ALL (ConsID) and INV_PROD_ALL (ProdID). When the producer and consumer are in different blocks, the first instruction writes back not just the local L1 but also the whole local block's L2 to the L3. Similarly, the INV_PROD_ALL instruction self-invalidates not only the local L1 but also the whole local block's L2.

A program annotated with WB_CONS and INV_PROD runs correctly both within a block and across blocks without modification.

2.4 Compiler Support

This work was done in collaboration with Sanket Tavarageri from Ohio State University. We present the details in the Appendix.

2.5 Experimental Setup

Since this thesis is about shared-memory programming, we evaluate the first programming model (Section 2.2) by running programs within a block, without any MPI component. We evaluate the second programming model (Section 2.3) by running programs across blocks.

Consider the intra-block runs first. We evaluate the configurations in the upper part of Table 2.2. For programming simplicity, the baseline uses WB_ALL and INV_ALL for all the synchronization primitives discussed in Section 2.2.1 and shown in Figure 2.3. Then, in *B+M*, *B+I*, and *B+M+I*, we augment the baseline with the MEB, IEB, and both MEB and IEB, respectively. These hardware buffers are only used in critical sections, which are small. Finally, we compare to the same machine with hardware cache coherence (*HCC*).

Intra-Block Experiments	
Name	Configuration
Base	Baseline: WB_ALL and INV_ALL
B+M	Base plus MEB
B+I	Base plus IEB
B+M+I	Base plus MEB and IEB
HCC	Hardware cache coherence
Inter-Block Experiments	
Name	Configuration
Base	Baseline: WB_ALL and INV_ALL to L3
Addr	WB and INV of addresses to L3
Addr+L	WB_CONS and INV_PROD

Table 2.2: Configurations evaluated.

We run the SPLASH2 applications, which use pointers heavily and have many types of synchronization. Specifically, we run FFT (64K points), LU (512x512 array, both contiguous and non-contiguous), Cholesky (tk15.O), Barnes (16K particles), Raytrace (teapot), Volrend (head), Ocean (258x258, both contiguous and non-contiguous), and Water (512 molecules, both nsquared and spatial). We use the SESC cycle-level execution-driven simulator to model the architecture in the upper part of Table 2.2. The architecture has 16

cores in a block. The coherent machine we compare to (*HCC*) uses a full-mapped directory-based MESI protocol.

Intra-Block Experiments	
Architecture	16 out-of-order 4-issue cores
ROB	176 entries
Private L1	32KB WB, 4-way, 2-cycle RT, 64B lines
Shared L2	2MB WB, 8-way/bank, 11-cycle RT (local bank), 64B lines
On-chip net	2D-mesh, 4 cycles/hop, 128-bit links
Off-chip mem	Connected to each chip corner, 150-cycle RT
Inter-Block Experiments	
Architecture	4 blocks of 8 cores each
Shared L3	16MB WB, 8-way/bank, 20-cycle RT (local bank), 64B lines

Table 2.3: Architecture modeled. RT means round trip.

We now consider the inter-block runs. We evaluate the configurations in the lower part of Table 2.2. The baseline is a simple design that communicates via the L3 cache. This means that a WB pushes the dirty lines to both L2 and L3, and an INV invalidates the lines in both L1 and L2. Moreover, it always uses WB_ALL and INV_ALL. The more optimized *Addr* configuration communicates via L3 but specifies the target addresses in the WB and INV. Finally, *Addr+L* uses our level-adaptive WB and INV instructions, which also specify the target addresses.

We run four OpenMP applications, namely EP, IS and CG from the NAS Parallel Benchmark suite, and a 2D Jacobi benchmark that we developed. CG is irregular. We use an analysis tool that we developed based on the ROSE compiler [43, 38] to instrument the codes with WB_CONS and INV_PROD instructions. We do not apply nested parallelism but, instead, only parallelize the outermost loop in a nest across all processors. We do not apply any loop optimization such as loop interchange or loop collapse either.

The architecture modeled is shown in the lower part of Table 2.2. We model 4 blocks of 8 cores each. The parameters not listed are the same as in the intra-block experiments.

2.5.1 Area analysis

An incoherent cache hierarchy requires less area that a hardware-coherent one, even if we neglect the coherence controller logic. Specifically, consider the incoherent hierarchy first. While we explained per-byte dirty bits for the finest granularity in previous sections, it is more practical to use per-word dirty bits for

many parallel programs, which have word or coarser granularity for inter-thread communication. We assume per-word dirty bits in the evaluation. The lines in the L1 and L2 caches need a single valid bit plus per-word dirty bits; the lines in the last level (L3) cache only need a valid and a dirty bit. Consider now a MESI protocol, which has 4 stable line states and many transient ones. It easily needs 4 bits to encode a state. Hence, each line in the L1, L2 and L3 caches needs 4 bits for the state. In addition, each line in the L2 and L3 caches needs directory information. For example, if we consider a full-mapped directory for the hierarchy in the inter-block experiments, each L2 line needs 8 presence bits (since we have 8 processors per block), and each L3 line needs 4 presence bits (since we have 4 blocks). Overall, the MESI protocol needs more area because it requires more bits in the L3 cache, which is the largest one, while the incoherent cache hierarchy adds the overhead mostly to the smaller L1 and L2 caches.

For a cache hierarchy with 4 blocks, we find that a full-map hardware-coherent cache hierarchy has an overhead of 3.65 Mb, while the incoherent one has an overhead of only 2.96 Mb, which is 18.9% less. With more blocks, the incoherent hierarchy has more scalability because it has a constant cost regardless of the number of blocks; the hardware-coherent one requires more area for the directory as we add more blocks. For example, with 8 blocks and 8 cores per block, the hardware-coherent hierarchy has an overhead of 9.34Mb, while the incoherent one has an overhead of 5.92Mb, which is 36.6% less.

2.6 Evaluation

This section evaluates our proposed hardware support and programming models for an incoherent hierarchical cache hierarchy like the one in Runnemede [14]. The evaluation has two parts. In Section 2.6.1, we evaluate our intra-block hardware and programming model. We compare the performance to that of the same multicore with hardware cache coherence.

In Section 2.6.2, we evaluate our inter-block hardware and programming model. For this case, there is no standard hierarchical hardware cache coherence that we can compare to. We would have to design one such protocol, explain it and justify it. Therefore, instead, we focus on evaluating the effect of our level-adaptive WB and INV instructions. Hence, we compare the performance to systems with WB and INV that always communicate via the last-level cache.

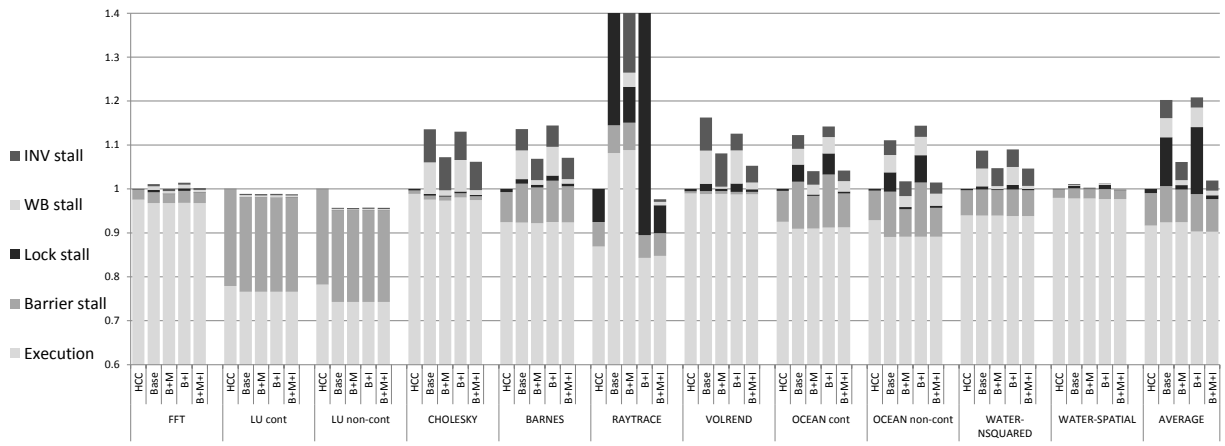

Figure 2.8: Normalized execution time of the applications.

2.6.1 Performance of Intra-block Executions

Figure 2.8 shows the execution time of the applications for the different architectures considered. From left to right, the bars correspond to *HCC*, *Base*, *B+M*, *B+I*, and *B+M+I*. For a given application, the bars are normalized to *HCC*. Each bar is broken down into 5 categories: stall time due to INV (*INV stall*) and WB (*WB stall*), stall time due to lock (*lock stall*) and barrier (*barrier stall*) synchronization, and rest of the execution. Barrier stall time mostly stems from load imbalance between threads. Lock stall time is the time spent waiting for lock acquires. Note any stall time for fine-grained INV/WB instruction is included in the execution time.

We classify the applications into those that exhibit coarse-grain or relatively little synchronization (FFT, LU cont, LU non-cont, and Water Spatial) and the rest. In the former class, the WB and INV overheads have very little impact. As a result, all the architectures considered have performance similar to *HCC*.

The rest of the applications have finer-grain synchronization. In these codes, the overhead of WB and INV stalls in *Base* is often large and, in the case of Raytrace, very large. In Raytrace, where the *Base* bar reaches over 2, there are frequent lock accesses in a set of job queues. Its fine-grain structure is the reason for the large overhead. The bars also show that the overhead in *Base* comes mostly from *INV stall*, *WB stall*, and *lock stall*. The latter is largely the result of *WB stall* in the critical section and its chaining effect to other acquire waiters. On average for all the applications, *Base*'s execution time is 20% higher than *HCC*.

We can speed-up execution by using the MEB to reduce the number of WBs that are needed before exiting the critical sections. This optimization succeeds in eliminating practically all the *WB stall* and *lock*

stall. We can see, therefore, that the MEB is very effective. The resulting bars (*B+M*) are close to *HCC*. One exception is Raytrace, where the bar is still around 1.4.

The third bar (*B+I*) shows that the IEB alone is not very effective to speed-up execution. The *INV stall* reduces a bit, but the *WB stall* and *lock stall* return, and the bars return to about the same height as *Base*. The problem is that, even though the number of INVs decreases, there is a large number of WBs before exiting the critical sections, which lengthens the critical sections and slows down the program. The fewer misses due to fewer INVs do not help significantly. Moreover, the IEB is so small that it sometimes overflows, becoming ineffective.

However, the fourth bar (*B+M+I*) shows that, by adding both the MEB and the IEB, we get the best performance. All the categories of overheads decrease. As a result, the programs are now not much slower than with hardware cache coherence (*HCC*). On average, the execution time of the applications with *B+M+I* is only 2% higher than with *HCC*. This is a large speed-up compared to *Base*, whose execution time was 20% higher than *HCC*. Overall, we have a system without hardware cache coherence that is about as fast as one with hardware coherence.

We now consider the traffic generated by *HCC* and *B+M+I*. Given that *HCC* and *B+M+I* have a similar average performance, their relative traffic is a proxy for their relative energy consumption. Figure 2.9 shows, for each application, their relative traffic in number of flits normalized to *HCC*. Each bar is broken down into traffic between the L2 cache and memory (*memory*), and three sources of traffic between the processor and L1/L2: linefill due to read/write misses, writeback traffic, and invalidations. Traffic-heavy synchronization using benign data races is converted to hardware synchronization primitives.

From the figure, we see that these applications have on average slightly less traffic in *B+M+I* than in *HCC*. This is despite the fact that these applications were not written for an incoherent cache hierarchy. As a result, *B+M+I* suffers from the fact that the analysis of which words need to be invalidated or written back is not very precise, and often requires the use of INV ALL and WB ALL. On the other hand, *B+M+I* removes traffic over *HCC* in three ways. First, *B+M+I* causes no invalidation traffic. Second, *B+M+I* does not suffer from ping-pong traffic due to false sharing. Finally, *B+M+I* performs writeback of only dirty words, thanks to its per-word dirty bits. Overall, *B+M+I* causes only slightly less traffic and, hence, consumes about the same energy as *HCC*.

We expect that, for applications written for an incoherent cache hierarchy, the total traffic and energy

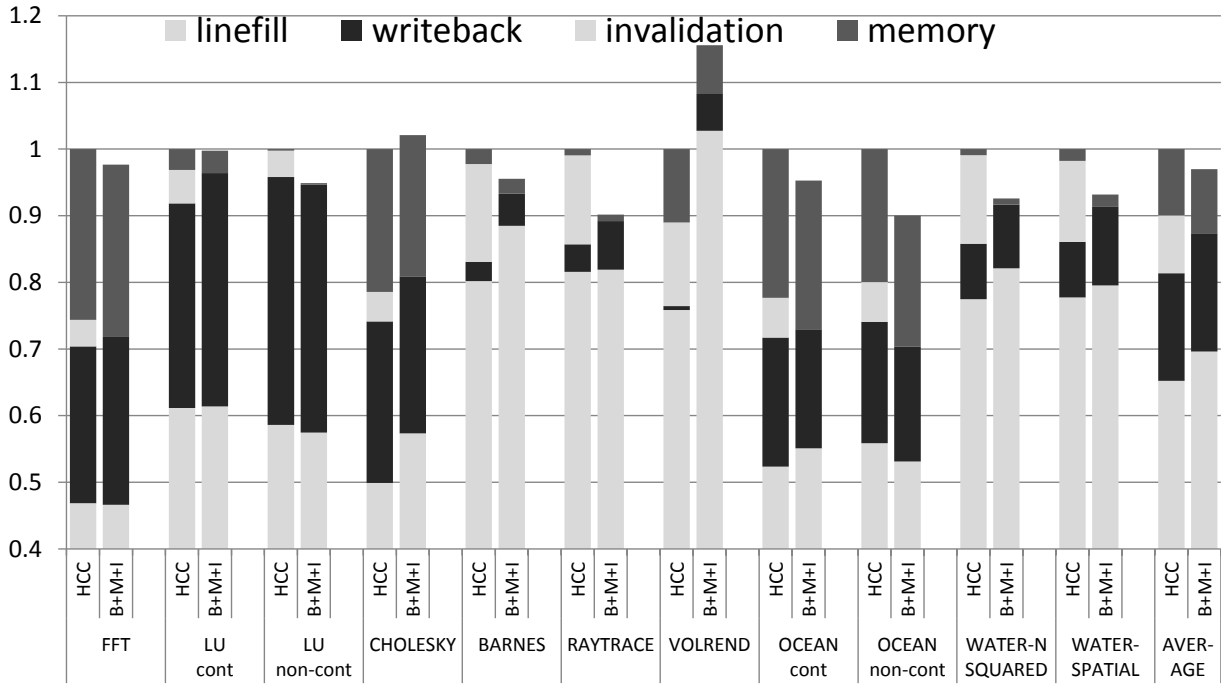

Figure 2.9: Normalized traffic of the applications.

in *B+M+I* will be lower than in *HCC* — because of the three reasons listed above. Moreover, even if the performance and energy is similar for *HCC* and *B+M+I*, the latter has the major advantage of needing simpler hardware, which translates into lower time-to market for the machine, and lower machine cost.

2.6.2 Performance of Inter-block Executions

To evaluate the effect of level-adaptive WB and INV, we start by counting the number of global WBs (those gone to the L3) and global INVs (those gone to the L2). Recall that *Addr* always performs global WBs and INVs, while *Addr+L* only performs them if producer and consumer are in different blocks. Figure 2.10 compares these counts for *Addr* and *Addr+L* for our applications. The bars are normalized to the values for *Addr*.

Only Jacobi and CG take advantage of level-adaptive instructions (*Addr+L* bars). In Jacobi, *Addr+L* reduces the number of global WBs and INVs to one third of the number in *Addr*. This shows the impact of WB_CONS and INV_PROD. In CG, as explained in Section 2.3.2, we use INV_PROD but not WB_CONS. Hence, *Addr+L* only reduces the number of global INVs, not WBs. The number of global INVs remaining is 78% of the number in *Addr*. With these level-adaptive instructions, we reduce network traffic and miss

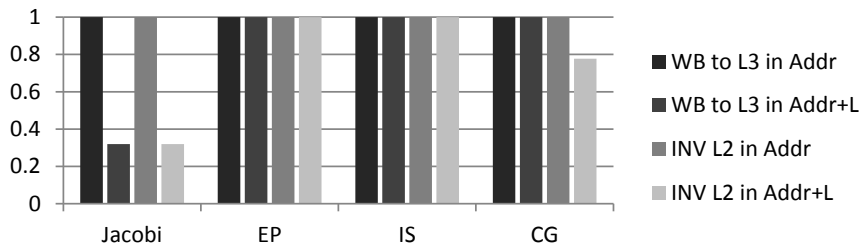

Figure 2.10: Normalized number of global WB and INV.

stall time.

However, EP and IS show no impact. In these applications, the major data communication pattern is reductions. Since a reduction does not have ordering, it is not possible to know the producer-consumer order. Thus, level-adaptive WB and INV cannot help. To exploit local communication, one could re-write the code to have hierarchical reductions, which reduce first inside the block and then globally.

Figure 2.11 compares the performance of *Base*, *Addr*, and *Addr+L* for the applications. The bars are normalized to *Base*, which always performs WB ALL to L3 and INV ALL to L2. We see that, in Jacobi, knowing which addresses to WB and INV pays off (*Addr*). The remaining WBs and INVs are relatively few and, therefore, making some of them local with *Addr+L* has only a modest further impact. Overall, the execution time of *Addr+L* is 36% of *Base*. In CG, *Addr+L* has a significant impact. Its reduction in global INVs attains an execution speed-up of 15% over *Base*. EP and IS do not show gains.

Overall, we conclude that level-adaptive WB and INV can speed-up codes by transforming some global operations into local. Their impact is a function of the contribution of the global operations to the execution time to start with. On average for our applications, *Addr+L* reduces the execution time by 5% over *Addr* and by 20% over *Base*.

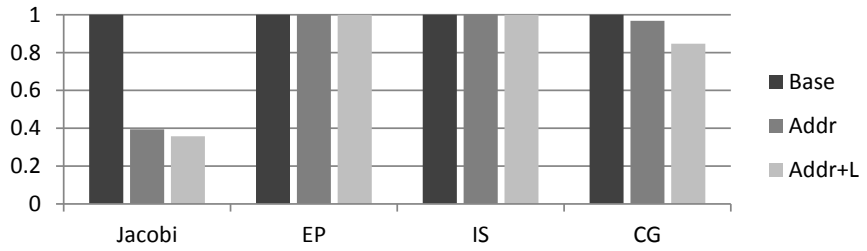

Figure 2.11: Normalized execution time of the applications.

Chapter 3

Evaluating Scratchpads in Traleika Glacier

Exascale system design brings many challenges in all disciplines. An unprecedented large problem size and system size require keeping scalability in mind throughout all design phases. Requirements for both performance and energy efficiency lead to fundamental rethinking from the ground up. Hardware needs to be re-examined from every aspect; and software needs many changes for maximum concurrency and minimum data movement. As a result, the Traleika Glacier (TG) project (X-Stack) [31] adopts near-threshold voltage for energy efficiency, narrow-issue simpler core design, dataflow-like parallel programming language, and explicit control over data movement.

Complicated requirements of exascale system drive us in two directions. First, HW/SW co-design is needed. Second, multi-scale analysis is required. While system level scalability can be explored with new software infrastructure on currently available large systems, fine-grained analysis complements it with detailed analysis of unit performance and energy efficiency on the target hardware. Our focus of evaluation is on fine-grained analysis with simulation of the target hardware architecture.

For HW/SW co-design and fine-grained evaluation, architectural simulation is key. Simulation enables design space exploration. On the hardware side, many parameters such as issue width, register file size, on-chip memory capacity, and network configuration can be configured. Simulation provides easy ways to evaluate the effect of parameter changes or architectural changes without actual hardware implementation. Software also gets benefits by running a full stack software on the simulator. Thus, software can be implemented and evaluated before actual hardware implementation or FPGA prototyping.

In this chapter, we evaluate through simulations the TG chip using scratchpads only. Our simulations include the whole system stack, which has been developed under the DOE X-Stack project. The software stack includes a dataflow-inspired programming model where programmers break computations into event-Driven Tasks (EDTs) that use datablocks (DBs), a runtime system that schedules such tasks called open community runtime (OCR), and an llvm-based compiler.

Our key observation from the evaluation is the following:

- In an architecture with as much on-chip parallelism as TG, it is important to schedule the event-driven tasks(EDTs) for parallelism, so that individual EDTs execute as soon as possible, subject to the data dependencies of the program. Typically, at any given time, there are many idle XEs that can be utilized.

- Given the cluster-based, hierarchical organization of on-chip memory modules, it is key to ensure that the datablocks (DBs) are close to the EDTs that use them. This means that we need DB allocation and migration for locality. Moreover, when the DB is not needed anymore, de-allocation to free-up space for other DBs is beneficial.

- Intelligent DB allocation is a requirement not just to attain good performance; it is a requirement for energy savings. It makes a big difference in energy consumption whether or not the DB and its EDT are located close by.

- It is hard to manually perform EDT scheduling for parallelism and DB management for locality. The programmer has to understand the code well. It is still an open problem regarding whether or not these operations can be effectively performed automatically.

- The system evaluated still has certain overheads and inefficiencies that need to be improved in future versions. In particular, launching and migrating EDTs is slow. In addition, some of the statistics do not provide correct insights due to the limitation of the current simulator.

In this chapter, we introduce key features of our target platform used in the X-Stack project in Section 3.1. Benchmark kernel programs are explained in Section 3.2, and detailed performance/energy efficiency evaluation follows in Section 3.3. We also discuss the limitation of current studies regarding the existing simulator framework. We conclude with our suggestion for future directions of exascale research platform in Section 3.4.

3.1 Background for XStack

3.1.1 Hardware architecture

The TG project borrowed its processor design from a previous research prototype processor architecture used in the UHPC program [1]. The processor architecture is not user friendly but focused on providing potential energy efficiency. The compiler or runtime is expected to orchestrate data movement to minimize energy consumption. The architecture is also introduced as codename 'Runnemede' [14].

Hardware architecture of Runnemede has the following features:

First, Runnemede has a many-core architecture with two different types of cores. The Control Engine (CE) runs task-based execution runtime, which provides OS functionality (scheduling, memory management). The Execution Engine (XE) runs tasks of user applications. The CE schedules an executable task on an available XE, and the XE runs the scheduled task until the completion of the task. These two types of cores are optimized for different goals. The CE is a full-featured core to handle OS and run legacy codes of the standard library while the XE is geared to low-power processor using RISC instruction set, FP-specific ISA, large register files, simple pipeline, and no interrupt handling.

Second, Runnemede has hierarchical structure. A single CE controls scheduling of multiple XEs. One CE and multiple XEs, for instance 8 XEs, establish a block. A block has block-shared memory, and network interface to other components in the chip. A unit consists of multiple blocks with unit shared memory and another network interface. A chip consists of multiple units with chip shared memory. This hierarchical structure allows designers to make a many-core chip easily using a core/block as a building block.

Third, Runnemede has scratchpad memories at every level of hierarchy as shown in Figure 3.1. Exploiting data locality of each level of hierarchy greatly contributes to minimum data movement and energy efficiency. While Runnemede architecture also has incoherent caches, the original Runnemede architecture uses scratchpad memory as main components of memory hierarchy. The current simulator lacks support for L1 incoherent caches; our evaluation in this chapter precludes L1 caches.

3.1.2 Software Platform

Open Community Runtime (OCR) [2] runs on the Runnemede processor and controls concurrency of user applications. OCR defines a dataflow-like programming model, provides low-level APIs to programmers

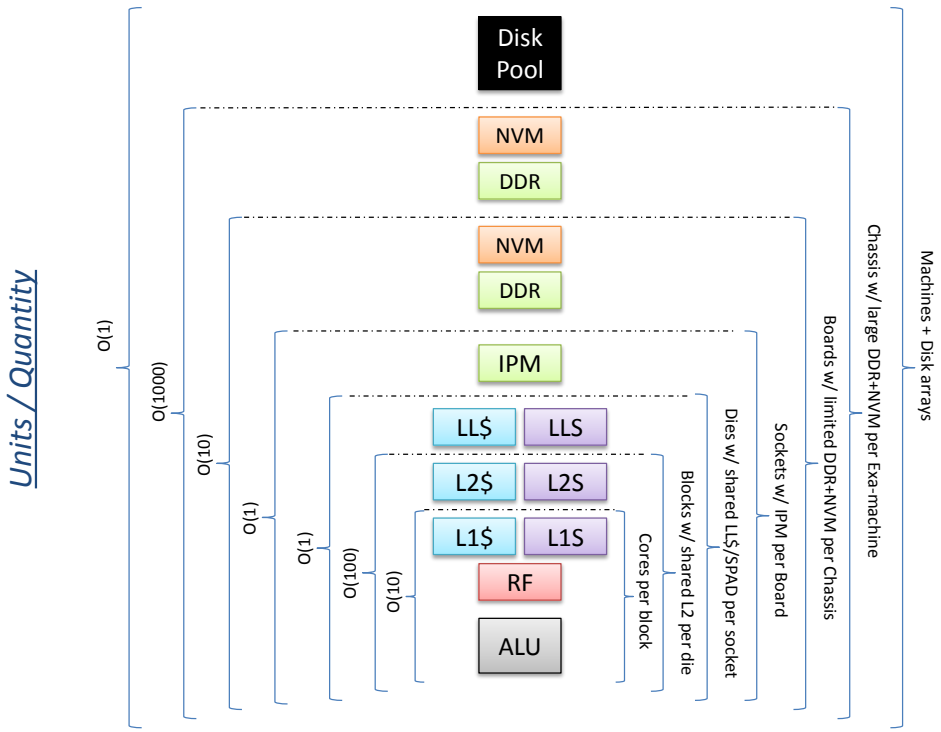

Figure 3.1: Hierarchical memory configuration (borrowed from the final report of [31]).

or compilers, and manages user applications and system resources for parallelism, data locality, resource management, resiliency, and energy efficiency.

In the OCR parallel programming paradigm, user applications consist of event-driven tasks (EDTs) and events between EDTs. An EDT is a portion of code that needs to be executed sequentially. Different EDTs can be run in parallel when there is no dependence between them. EDTs communicate with each other using a datablock (DB). A DB is a fixed-size contiguous memory, which is a unit for memory allocation/de-allocation/communication. When a DB is passed to another consumer EDT, the globally unique identifier (GUID) of the DB is passed.

An EDT has pre-slots for starting its execution and post-slots for notifying its completion to other EDTs. When all DBs in pre-slots are available, and events in pre-slots are satisfied, then the EDT is ready to be scheduled. The EDT releases DB and satisfies events to inform the next EDTs. Dependencies between EDTs are described by the programmer or compiler, and the OCR runtime is responsible for the scheduling of EDTs. Programming concepts and examples are shown in Figure 3.2. The above example is for producer-consumer, and the second example is for spawning two EDTs after the completion of the first EDT.

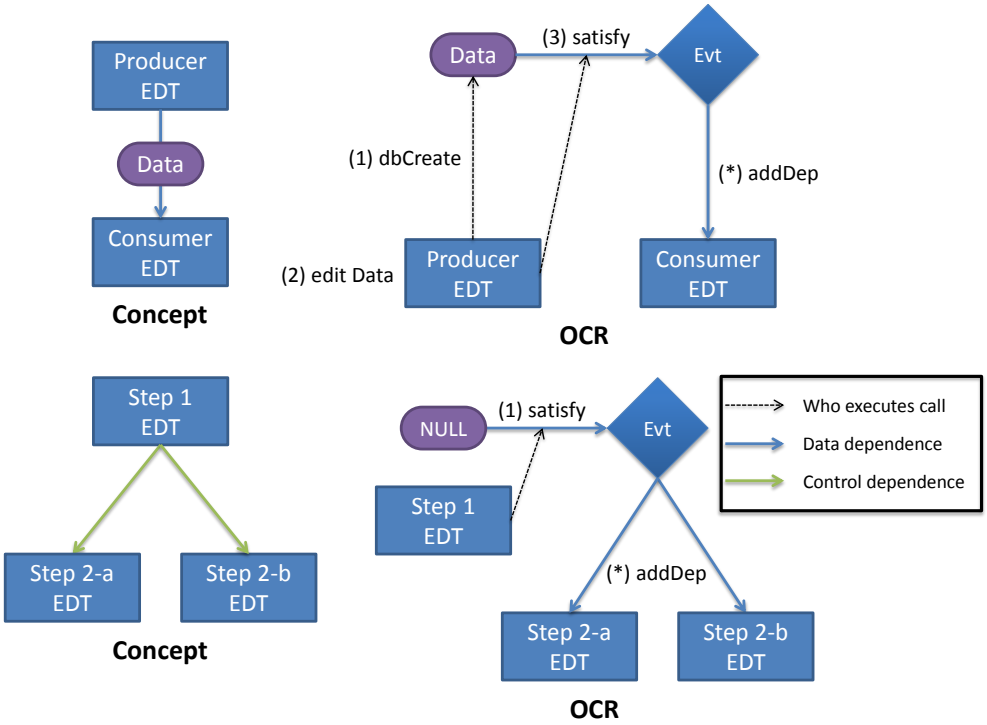

Figure 3.2: OCR dependency examples (borrowed from [31]).

On Runnemede architecture, OCR runs on CE. CE schedules ready-to-be-scheduled EDTs on available XEs. On the completion of the EDT, XE signals to CE. Then, CE tries to schedule a new EDT if there is an available EDT in the queue. During the execution of the EDT on XEs, the EDT can call the runtime for system functions.

Major differences from conventional parallel programs are two-fold. First, the OCR programming paradigm provides point-to-point synchronization. Data dependencies and events for control dependencies are knobs for point-to-point synchronization between EDTs. Compared with conventional synchronization such as barriers, this synchronization mechanism can allow more concurrency and less load imbalance by exposing more executable EDTs. Second, data are managed at DB granularity. Although one DB can be read/written by multiple EDTs concurrently, racy accesses from different EDTs are prohibited. A well-written program has a good partitioning of data into DB tiles, and DBs are associated with specific EDTs. This relation can be exploited to minimize data movement. For instance, scheduling relevant EDTs on the same XEs can benefit from data locality. Since memory allocation goes through memory hierarchy from L1 scratchpad memory through its block memory to DRAM, scheduling a consumer EDT on the XE which

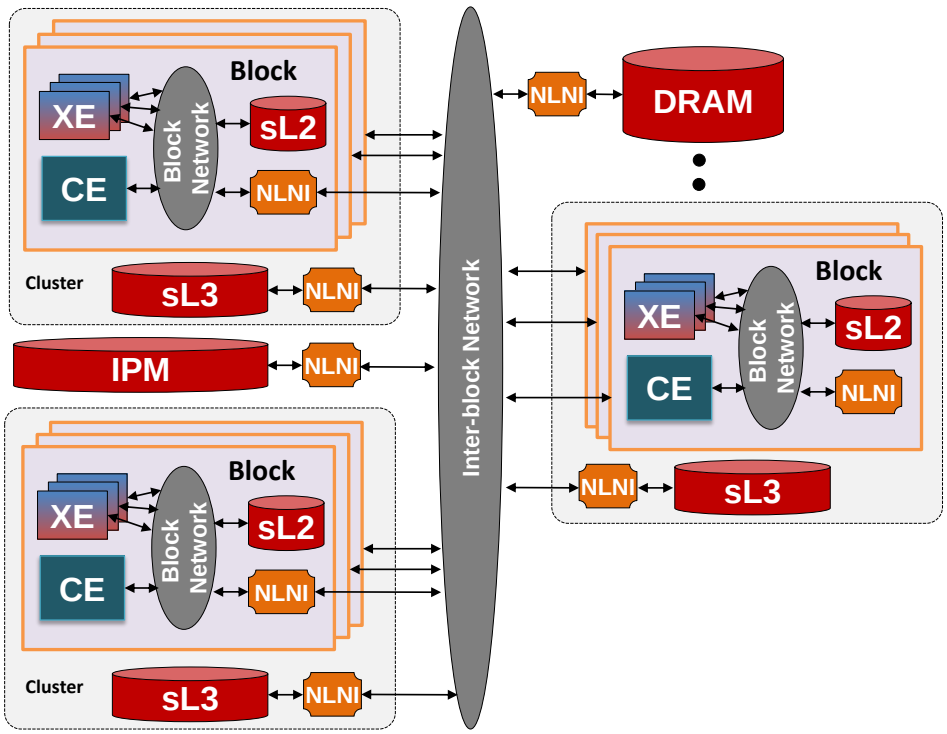

Figure 3.3: FSim architecture model (borrowed from the final report of [31]).

executed the producer EDT is a good heuristic for data locality.

3.1.3 Simulator

A functional simulator (FSim) [19] is provided to evaluate exascale system architecture. FSim was first developed under the UHPC program to provide a simulation environment for the system described in Section 3.1.1. It is adopted by the TG project as an abstraction of the hardware layer. FSim is highly parallel and scalable to accommodate many-cores as depicted in Figure 3.3, leading to reasonable evaluation time for HW/SW co-development. Parallel speed-up is achieved by running multiple threads for a multi-core block, and having multiple processes over multiple machines for multi-block configuration. While multi-threaded processes communicates using shared memory, multiple processes communicate through sockets. Next-level network interface (NLNI in Figure 3.3 connects processes, which is an equivalent component to a network switch in an on-chip network. In this way, FSim is geared to today's cluster systems with multi-core processors.

Since the initial motivation of the simulator is to have a platform for quick design exploration and

to enable SW development on evolving hardware specification, FSim did not have the requirements for a detailed timing simulation. Instead, FSim models functionality behaviorally and provides images of system components. For instance, the simulation agent for CE is implemented using QEMU [3], which is a machine emulator, because functional emulation is sufficient to evaluate correctness of the SW stack. QEMU also provides a good set of full system images, which enables CE to function as a general purpose core running runtime software. The simulation agent for XE is based on a configurable processor model, which has basic modeling for latency of functional units, dependency between instructions, and instruction-issuing width. Shared memory (block shared memory, unit shared memory, chip shared memory), DRAM, and networks are provided as rough functional models. They can be enhanced for detailed modeling since they are provided as plug-ins.

While FSim provides scalable, reasonable simulation models for SW design, its support for the hardware architect is relatively lacking. The first limitation is its timing model. As a timing model of the system, a Lamport logical clock scheme [37] is applied. CE and XE increment their Lamport clocks at every instruction execution, and communication between them is synchronized using the SYNC function. Since *receive* is causally related to *send*, the message sent has an incremented Lamport clock value. At the receiver, a new Lamport clock value is determined as the maximum of the local Lamport clock and the received Lamport clock. Passive agents such as shared memories also update their local Lamport clocks to preserve the causality of accessing memories from different agents. Figure 3.4 shows behavior of the Lamport clock in FSim.

```
L is local Lamport clock.
M is Lamport clock value in the message.

agents:
        execute: INC(L)
        send: M = INC(L)
        receive: L = SYNC(M) = max(L, M)
```

Figure 3.4: FSim Timing Synchronization.

While the Lamport clock preserves happens-before order between events of cores, the reliance of synchronization solely on message communication leads to high variance of the resulting timestamp. FSim does not have throttling of active agents; thus, the local Lamport clock of agents can differ significantly. Under this circumstance, any direct/indirect communication between agents can incur *clock warping* for

slowly-progressing agents. When a simulation thread for XE1 gets more processor time, or executed instructions on XE1 are simpler and quicker than those on XE2, then XE1 can have the bigger timestamp. When XE2 accesses XE1 or any XE1's touched agent, XE2's Lamport clock must be synchronized with the bigger value. This makes the current Lamport clock scheme inappropriate to be used as a performance number. We reduce the variance and clock warping effect by eliminating unnecessary timestamp synchronization between XE and CE and prohibiting clock warping through shared memory agents. While this new scheme provides a good sense of proxy numbers, it still has limitations in accuracy; reported performance numbers should be understood as proxy numbers and relative numbers between different configurations.

For energy estimations, FSim provides a dynamic capacitance model (C_{dyn} model). From a basic power model, $P = afCV^2$, the circuit activity factor a and the total switched capacitance C are variable while the operating frequency f and the operating voltage V are fixed. Thus, dynamic capacitance calculation focuses on aC, which is called C_{dyn}. Each instruction has its own capacitance number and activity factor, which are results of an instruction fetch, use of functional units, memory accesses, etc. FSim gets the configuration value of each component capacitance, and sums them up for total dynamic capacitance. The resulting C_{dyn} is used for core power/energy calculation. Memory and network energy calculation is simpler than core energy calculation. FSim counts activity numbers and they are multiplied by unit energy for network/memory access.

3.2 Target Benchmark Program Study

We have used four kernel programs from OCR repository [4]: Cholesky, Smith-Waterman, FFT, and SAR. These four programs show distinct characteristics in degree of parallelism, memory access patterns, and data partitioning. The level of effort in programming also varies depending on programmers' efforts and understanding of OCR and TG architecture.

3.2.1 Cholesky

Cholesky is a decomposition method of a positive definite matrix into a product of a lower triangular matrix and its conjugate transpose. The algorithm is iterative, and values produced in the previous iteration are used in the current iteration. The algorithm has O(N^3) complexity because it iterates N (the matrix size in one dimension) times for matrix elements ($N \times N$). The simplified sequential algorithm is written in Figure 3.5.

```
for (k = 0; k < N; k++) {
  A[k][k] = sqrt(A[k][k]);
   for (j = k+1; j < N; j++) {
       A[j][k] = A[j][k] / A[k][k];
       for (i = k+1; i < N; i++) {
           A[i][j] = A[i][j] - A[i][k] * A[j][k];
       }
   }
}
```

Figure 3.5: Cholesky algorithm.

The parallel version of Cholesky divides an entire input matrix into $t \times t$ tiles, where t (the number of tiles in one dimension) is N (the matrix size in one dimension) divided by T (the tile size in one dimension). An external program transforms an original matrix into a tiled lower triangular form, and the Cholesky program starts from it.

The original sequential algorithm has three different tasks. In Computation A, it computes the square root of the topmost element in each iteration. In Computation B, for the leftmost column, each element is divided by the topmost element. Finally, in Computation C, for all other remaining elements, two elements from the leftmost column are operated on. After tiling, each tile has a different task set depending on its location. The topmost tiles need to have all three computations (Sequential EDT type), while the leftmost column tile has computations B and C (Trisolve EDT type). The remaining tiles need to have computation C only (Diagonal/non-diagonal EDT type). Thus, three different EDTs are used for the different types of tiles.

Figure 3.6 shows three EDT types and their read set. An EDT for the current tile (blue) has dependencies on EDTs for tiles being read (green). The top row shows the EDTs executed in iteration i; the second row shows the EDTs executed in iteration i+1. In each iteration, there are three types of EDTs. The Sequential EDT is the single tile in blue in the leftmost plot of each iteration. It needs to execute first. The Trisolve Task EDTs are the blue tiles in the central plot of each iteration. They can only execute after the Sequential Task EDT completes. Finally, the Update Non-Diagonal and Update Diagonal Task EDTs are the blue tiles in the rightmost plot of each iteration. They can only execute after the Trisolve Task EDTs complete.

The total number of EDTs is calculated as follows:

The sum of tiles at iteration i is calculated as $\sum_{k=1}^{i} k$.

For all iterations, it is $\sum_{k=1}^{t} \frac{k(k+1)}{2} = \frac{1}{6}t^3 + \frac{1}{2}t^2 + \frac{1}{3}t$.

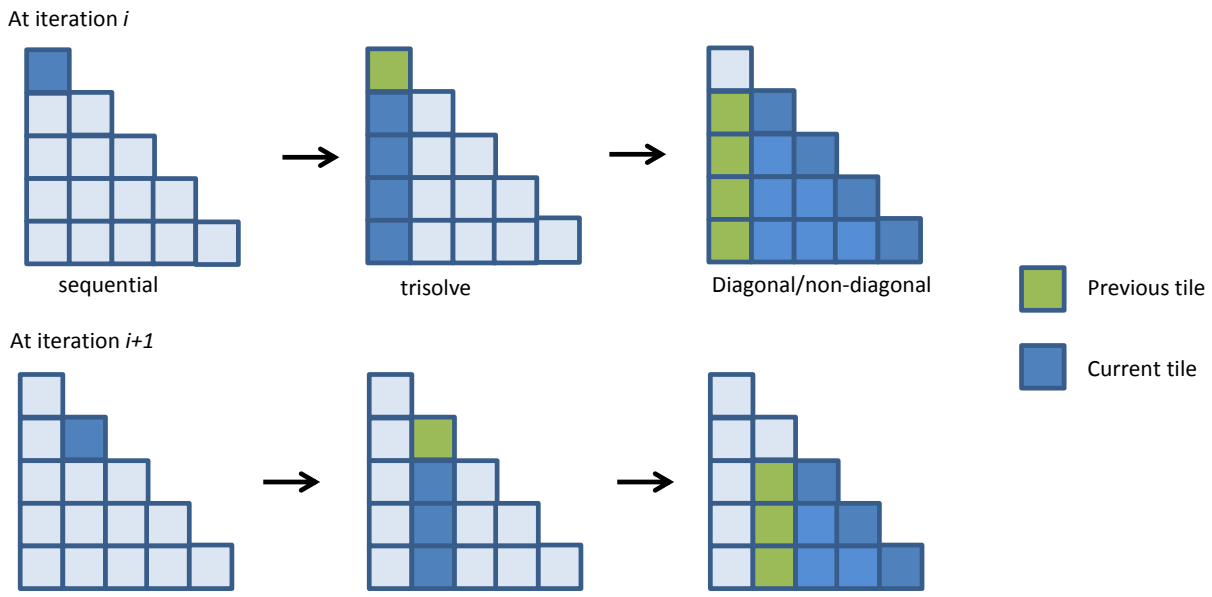

Figure 3.6: Dependencies between EDTs in Cholesky factorization over iterations.

For example, for tile number 10×10, the total number of EDTs is 220.

3.2.2 Smith-Waterman

Smith-Waterman is a DNA sequence alignment algorithm. The algorithm traverses an input data matrix from the top-left element to the bottom-right element with diagonal steps due to data dependence on the left element and the top element. In the parallel version, tiled data layout is used to avoid per-element synchronization.

Figure 3.7 shows the dependence structure between EDTs. The EDT for a tile has dependencies on the EDTs for the tiles in NW, W, N directions. In the figure, this is shown as the blue tile depending on the three green tiles. In turn, the tile processed in the current EDT creates data that is used in the EDTs for the tiles in the S, SE, E directions. This creates a wave-front style parallelism that leads to a maximum degree of concurrency of N when $N \times N$ tiles are available. Figure 3.8 shows concurrent execution of EDTs and scheduling with the conventional barrier synchronization.

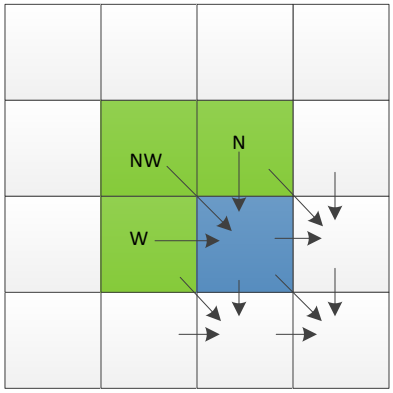

Figure 3.7: Dependence between EDTs in Smith-Waterman.

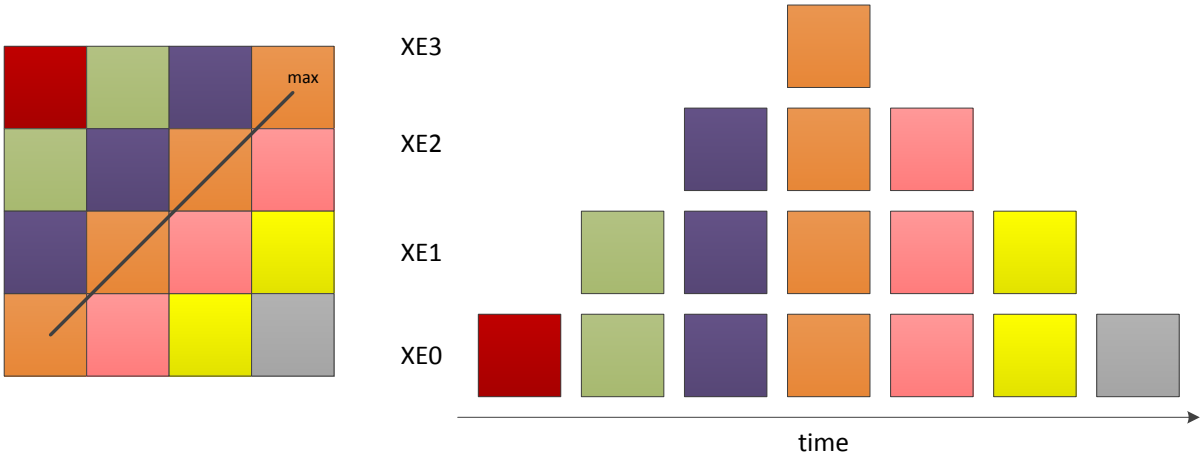

Figure 3.8: Concurrent execution of EDTs with diagonal parallelization and its expected scheduling on XEs.

3.2.3 FFT

FFT implements the Fast Fourier Transform algorithm by Cooley-Tukey. In the parallel version, the algorithm works in a divide-and-conquer style until each EDT gets the predefined minimum data size as depicted in Figure 3.9. However, this version of FFT does not use data tiling, thus only one DB is created and shared among all EDTs. This is a result of a direct translation from the sequential version of the program without thinking of the communication volume and the hierarchical memory configuration of TG architecture.

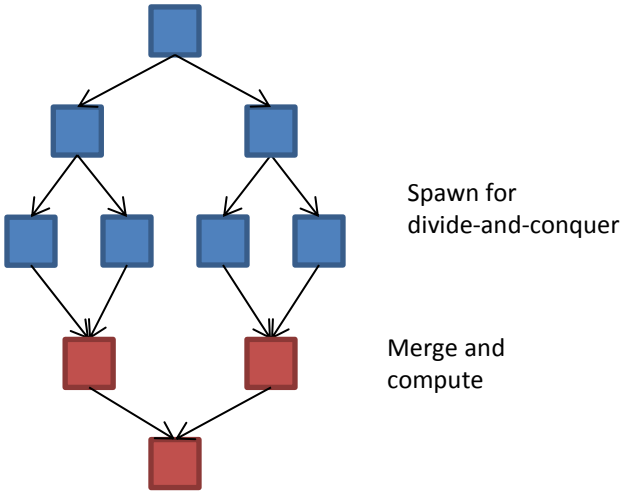

Figure 3.9: Dependence between EDTs in FFT.

3.2.4 SAR

SAR is a synthetic aperture radar algorithm, which is used to create images of an object, such as landscapes. An antenna, typically mounted on a moving platform such as an aircraft, forms a synthetic aperture and is used to capture multiple images. The application combines images from multiple antenna locations to create a finer resolution image.

The SAR application has 5 distinct phases of fork-join style parallelism as shown in Figure 3.10. During each phase, the application has a maximum concurrency of (Problem Size / Blocking Factor) tasks. The total number of tasks created during the entire execution is approximately 5 $\times$ (Problem Size / Blocking Factor). We experimented with a problem size of 64×64 pixels and a blocking factor of 32×32 pixels. Hence, we expect a maximum of 4 concurrent tasks and a launching task at any point in the program. We can expect low XE utilization with this concurrency level, and multi-block configuration suffers from this limitation severely.

3.2.5 Comparison

Table 3.1 shows a comparison of the four kernels in terms of parallelism and memory access behavior. Note that FFT has only a single DB allocated in mainEdt, which is a result of a direct translation from the sequential version of the program. Other programs have tiled data and associated EDTs.

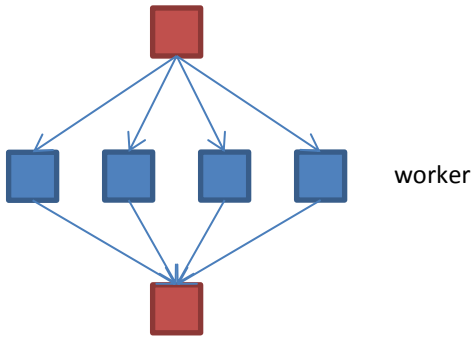

Figure 3.10: Dependence between EDTs in SAR.

Kernel	Parallelism Style	Degree of Parallelism	Number of Datablocks
Cholesky	Variable over time (decreasing)	max. $\frac{t(t-1)}{2}$	$\frac{t(t+1)}{2}$
Smith-Waterman	Variable over time (increasing in first half, decreasing in second half)	t	t^2
FFT	Variable over time (divide-and-conquer and merge)	$2^n/minimumprocessingsize$	1
SAR	Fork-join style	t^2	t^2

Table 3.1: Comparison of kernel programs (N is the problem size in one dimension, T is the tile size in one dimension, and t is N/T, which is the number of tiles in one dimension.)

3.3 Evaluation

We measure the execution time in number of simulation cycles, and energy consumption in FSim. While we try to take all four applications for the evaluation, Cholesky is extensively used to see effects of varying parameters since it is a well-written parallel program with proper tiling. We use 64KB L1 scratchpad, 1.25MB L2 block shared memory, and DRAM for memory allocation. These three levels are the maximum level that the current version of OCR memory allocator supports.

3.3.1 Performance Scaling

Figure 3.11 shows the execution time for different numbers of blocks and XEs per block, normalized to the execution time of the configuration with one block and one XE per block (b1x1). We use the b1x1 configuration as a baseline, which executes the EDTs sequentially.

Cholesky shows good performance scaling with more XEs up to b2x8 (two blocks with eight XEs each). Adding more XEs does not improve the performance for this problem size. Adding more blocks provides

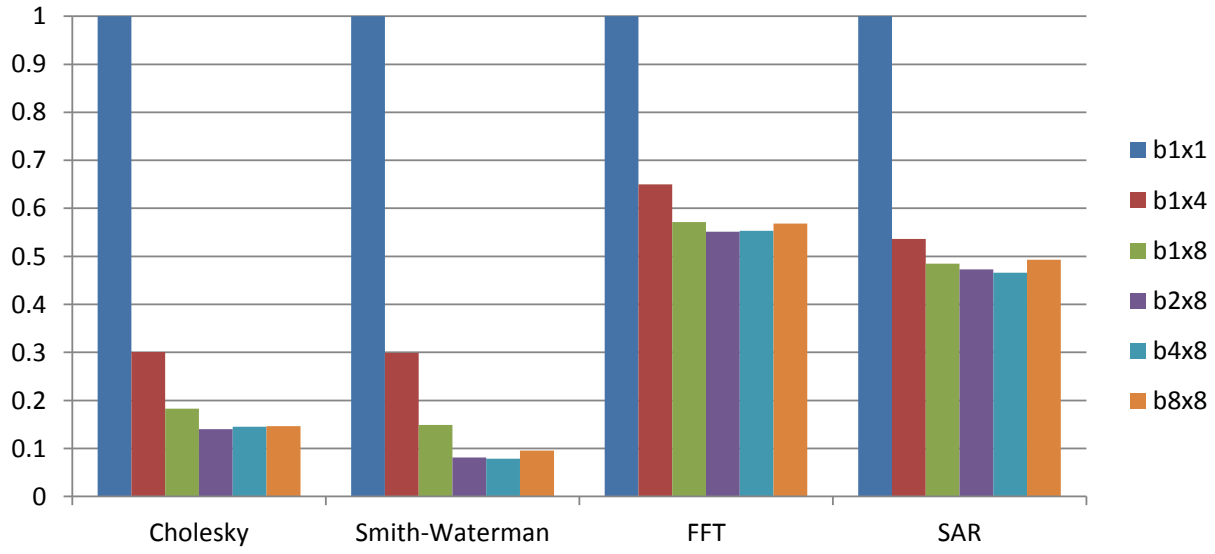

Figure 3.11: Normalized execution time of the applications.

more computation resources, but it also increases the latency of memory accesses if inter-block accesses are required. All runs use a tiled version of Cholesky with data size 500 and tile size 50. If we used a non-tiled version for the baseline, the baseline performance would be worse because the single data block would be allocated in the main memory, rather than on the block memories.

Smith-Waterman is run with data size 3200 and tile size 100. It shows performance improvement up to b2x8. FFT is run with problem size 18. It shows performance improvement until b1x8. More than 8 XEs do not help the performance much. SAR runs with the *tiny* problem size, and executes a 64×64 image size with 32×32 tile size. It shows meaningful speedups until b1x8. The limitation of parallel speedup comes from the limited concurrency level of the problem size.

3.3.2 Energy Consumption

Figure 3.12 shows the breakdown of the energy consumption with varying numbers of blocks and XEs per block. The bars are normalized to the one XE configuration (b1x1). We use the same problem sizes and configurations as in the previous section. Our basic expectation is that the energy consumption is approximately constant as we increase the number of XEs because the work done is similar (except for some overhead of parallelization and OCR). However, it is also possible that, with more blocks, the energy consumption decreases. This is because we have more block memory (BSM) to allocate DBs, which is

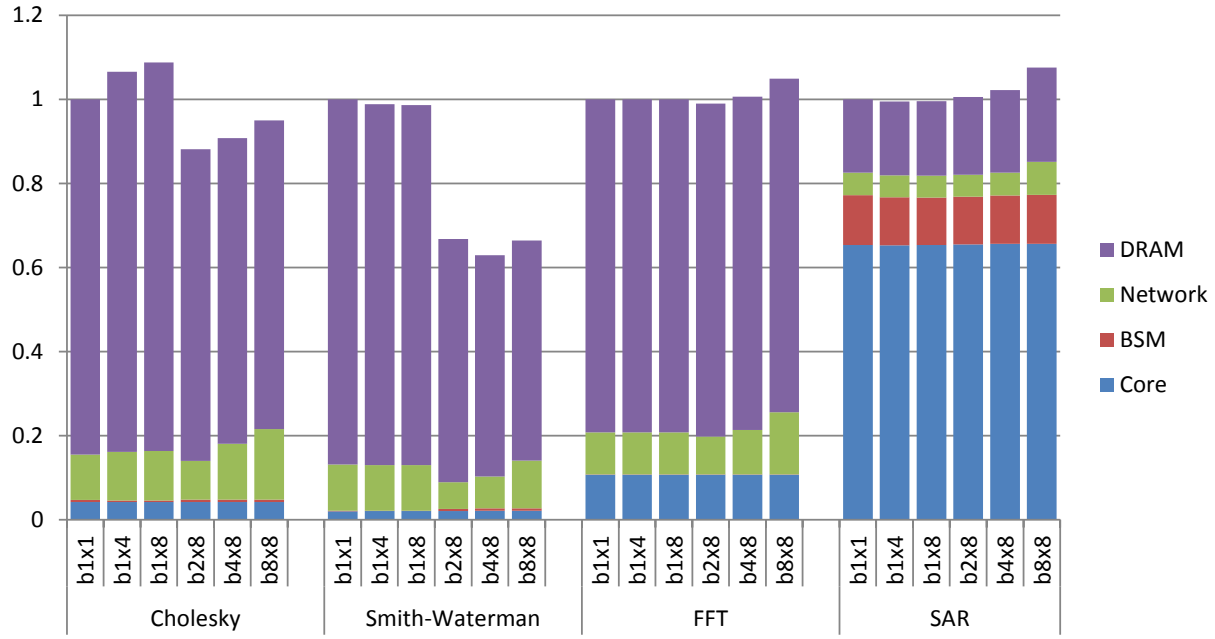

Figure 3.12: Energy consumption breakdown of the applications.

cheaper to access than DRAM memory.

From the figure, we see that DRAM energy consumption dominates in Cholesky, Smith-Waterman and FFT. This is because accesses to DRAM are very costly in energy. With a single block, the data size used in the programs is larger than the block memory size (1.25MB). Thus, even with tiling, not all DBs are located in on-chip memory; about half of them are allocated in DRAM. As we move to the b2x8 configuration, both Cholesky and Smith-Waterman show energy reduction. This is because of the increased block memory. More DBs are allocated in the block memories, and the number of DRAM accesses decreases. Unfortunately, FFT has a single DB, and it is large enough to be allocated in DRAM. Thus, most accesses go to DRAM. FFT shows the necessity of DB tiling. SAR is a more computation-intensive kernel, and its small problem size is absorbed by the block memory.

In general, with more XEs, the DRAM energy and network energy have a tendency to increase. This is because the scratchpad memory is used by both the program and OCR. The memory usage of OCR increases with more cores.

3.3.3 Effect of the Granularity of Parallelism

Varying the tile size results in two effects: changing the number of concurrent EDTs and the location of the memory allocation. Smaller tile sizes enable more concurrent EDTs; however, more EDTs/DBs can incur more storage overhead for OCR internal data structures. Larger tile sizes have trade-offs between parallelism and EDT overhead.

To see the effect of tiling, we use Cholesky with a matrix size of 500 and vary the number of tiles by changing the tile size from 25 to 250. The results are shown in Figure 3.13, normalized to b1x1 and tile size of 50. With bigger tiles, the parallelism is lower and, therefore, execution time is higher. With smaller tiles, we have more tiles and more parallelism. This reduces the execution time. In the case of tile size 25, we see speed-ups until the b4x8 configuration. However, for tile size 50, the execution time does not decrease after b2x8 because of EDT overhead.

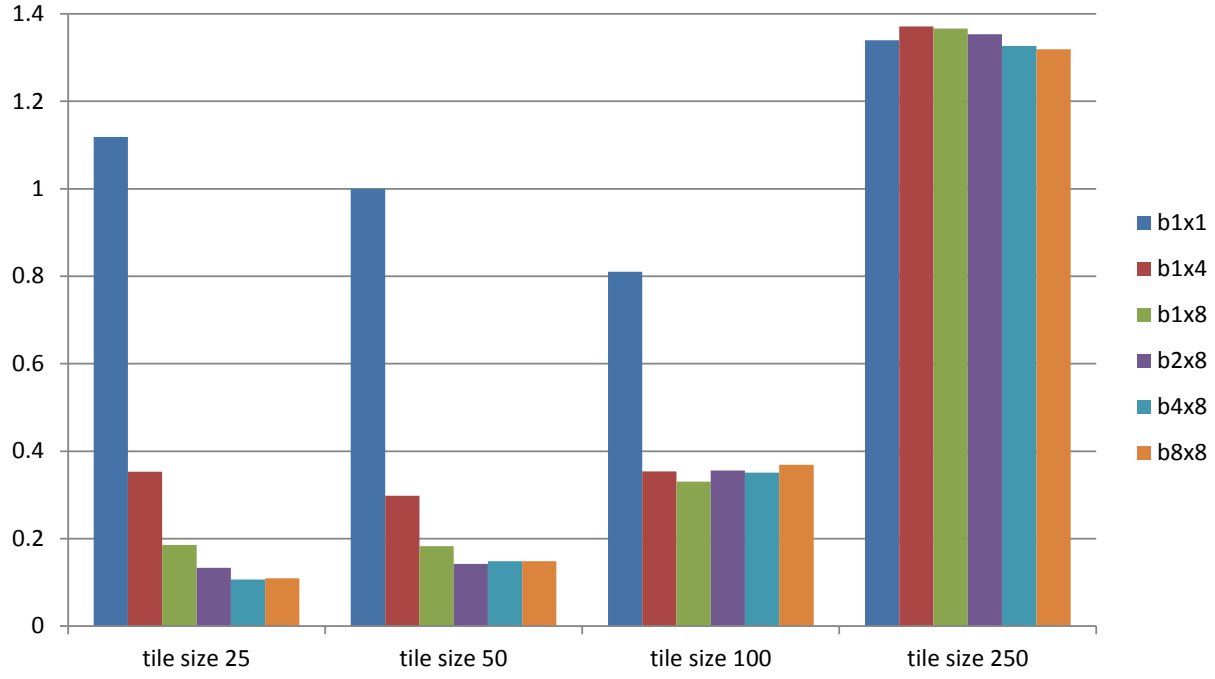

Figure 3.13: Normalized execution time with different tile sizes.

Going from tile size 50 to tile size 100, we reduce the total number of computing EDTs from 220 to 35. With tile size 100, the performance of b1x1 is better than the case of tile size 50 because of less overhead of EDTs. However, tile size 100 has lower scalability. Increasing the number of XEs hits scalability limits earlier than with tile size 25 or 50. In the case of tile size 250, there is no parallelism because all EDTs

processing tiles have a dependence chain. Therefore, increasing the number of XEs is useless.

Figure 3.14 shows the energy consumption with varying tile size. Tile size 100 shows the lowest energy due to less complexity of EDT relations. For smaller tile sizes, the memory allocation of more events and OCR data structures reduces the size of the block memory for the application.

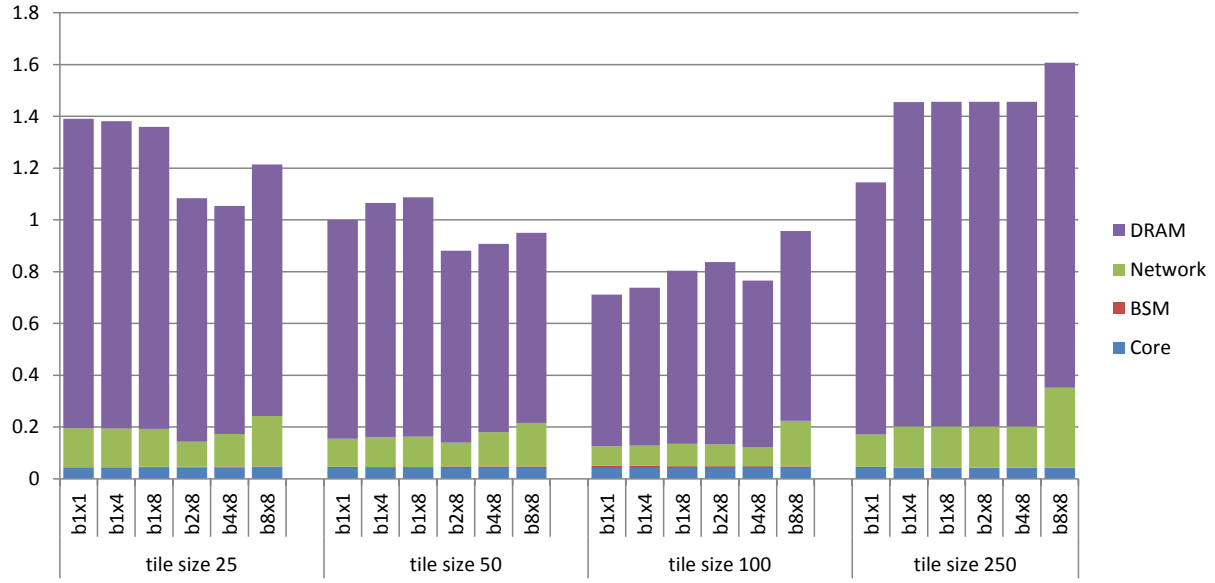

Figure 3.14: Normalized energy consumption breakdown with different tile sizes.

3.3.4 Scheduling EDTs for Parallelism

To see the impact of scheduling EDTs for parallelism, we consider Cholesky with the b1x4 configuration. In Figure 3.6, we showed the dependence between EDTs in Cholesky. With traditional barrier-based parallelization, iterations must be separated by barriers. Thus, all the EDTs in iteration i must be completed before any EDTs in iteration i+1 execute.

With OCR, EDTs from different iterations can be executed concurrently if there is no dependence remaining for the EDT. OCR tracks dependence between EDTs at finer-granularity. In Cholesky, the EDT of a tile in iteration i+1 does not need to wait for the EDTs of all the tiles in iteration i to complete. Instead, a tile in iteration i+1 can be processed as soon as all those it depends on (in iterations i and i+1) complete. This is what OCR enables in a manner that is transparent to the programmer. With OCR, we can overlap EDTs from different iterations, as long as the dependence between individual tiles are respected.

Figure 3.15 shows a timeline of the EDT scheduling observed during execution. The figure shows

which EDT is executing on each XE. The color code is as follows: the first gray block is the MainEDT, which creates all other EDTs, and sets dependence between them. Green blocks are Sequential Task EDTs, blue blocks are Trisolve Task EDTs, red blocks are Update Non-Diagonal Task EDTs, and purple blocks are Update Diagonal Task EDTs.

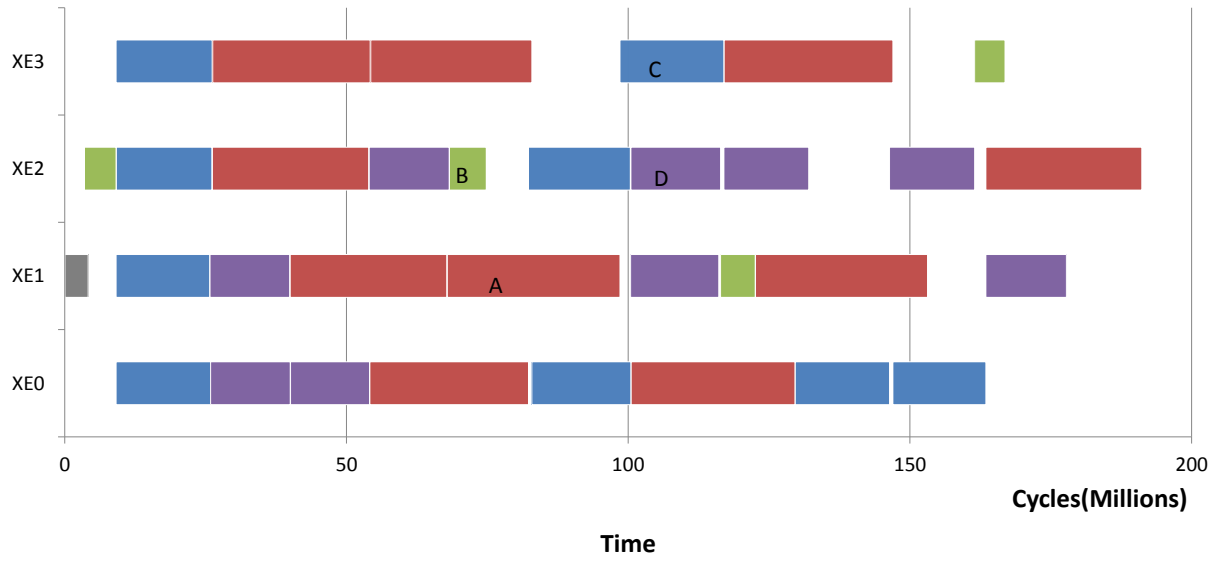

Figure 3.15: EDT scheduling timeline of Cholesky.

Figure 3.15 shows that EDTs do not wait for all the EDTs from the previous iteration to complete. In the figure, EDT A is an Update Non-Diagonal EDT from iteration 1, and EDT B, is a Sequential EDT in iteration 2. These two EDTs do not have any data dependence, even though they are in different iterations. Consequently, under OCR, they execute concurrently. The same occurs between independent EDTs in a single iteration. For example, EDT C is a Trisolve EDT from iteration 2 and EDT D is an Update Diagonal EDT from iteration 2. They happen not to have data dependence, and so OCR executes them concurrently. The result is more concurrency than under traditional barrier-based parallelization.

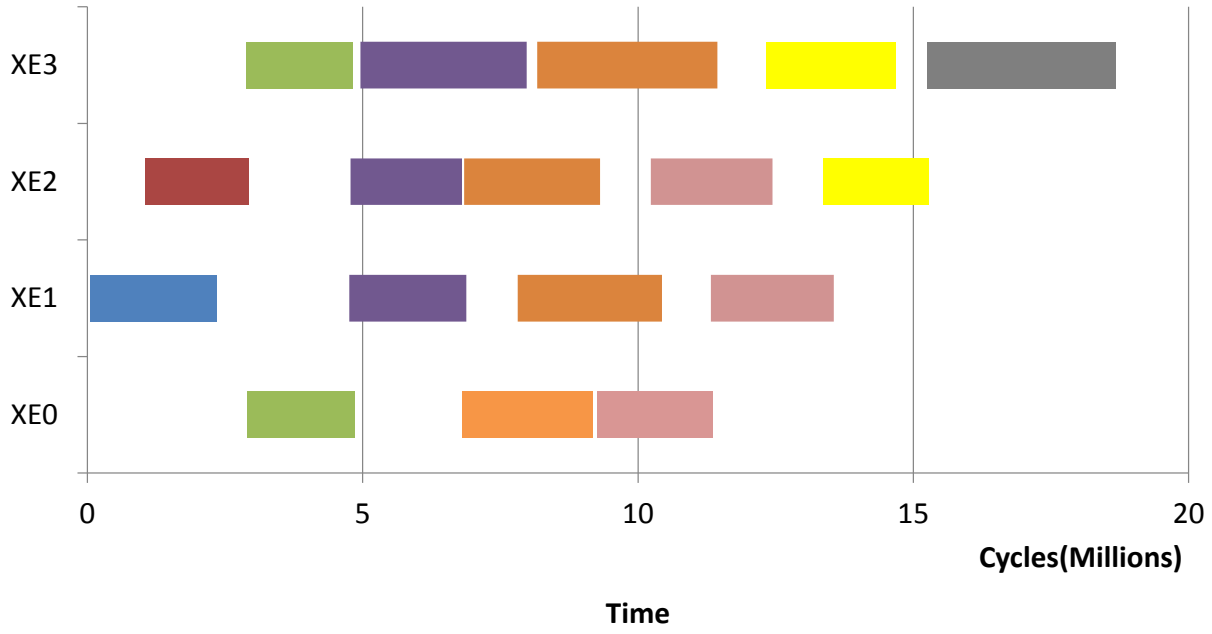

Figure 3.16: EDT scheduling timeline of Smith-Waterman.

Figure 3.16 shows a timeline of Smith-Waterman for 4x4 tiles running on 1 block with 4 XEs. We see that the execution is much more concurrent, and more XEs are busy on average. With global barriers, as in Figure 3.8, only tiles of the same color can run in parallel. In Figure 3.16, orange EDTs start execution before all the purple ones finish. Also, pink EDTs extend the time for which 4 XEs are running. With more overlapped execution, OCR enables better exploitation of fined-grained parallelism. The only effect that limits the concurrency is the true dependence between the EDTs.

Given this environment, we tried to improve the performance by using the hint framework, which schedules EDTs close to the memory module where the corresponding DBs are allocated. Programmers can give hints for scheduling, e.g. giving preference to the location of the DB in the first member in the dependence list. The effect of this hint-based scheduling should be reduced latency and energy; however, we did not find any noticeable benefits of using the hint framework. We observed that when all the XEs are busy, the scheduler chooses an available XE regardless of the hint. Moreover, when DBs are allocated in the DRAM, the EDT always needs to access DRAM, irrespective of where it is scheduled. Finally, it is hard for programmers to add hints for an EDT with multiple dependencies.

3.3.5 Multi-level Memory Allocation

In this section, we show the impact of managing the allocation and de-allocation of DBs for data locality. Exploiting locality leads to reduced memory access latency, and reduced energy in memory and the network. To illustrate this optimization, we apply it to Cholesky.

Cholesky has three distinct DB allocation places in the code (via ocrDbCreate). Initially, it allocates DBs for input data. During execution, both sequential EDTs and trisolve EDTs allocate DBs. The problem size considered is 500 with tile size 50, and we run it on a b1x4 configuration. In this case, the program requires more storage than the capacity of the L2 memory in the block (i.e., the BSM memory). Thus, after the BSM is filled with DBs, all subsequent DB allocation goes to DRAM. Figure 3.17 shows DB allocation result of b1x4 execution with data size 500 and tile size 50. The figure only shows DB allocation results of sequential EDT and trisolve EDT. Red block means DB is allocated in DRAM. All DBs are allocated in DRAM in this execution.

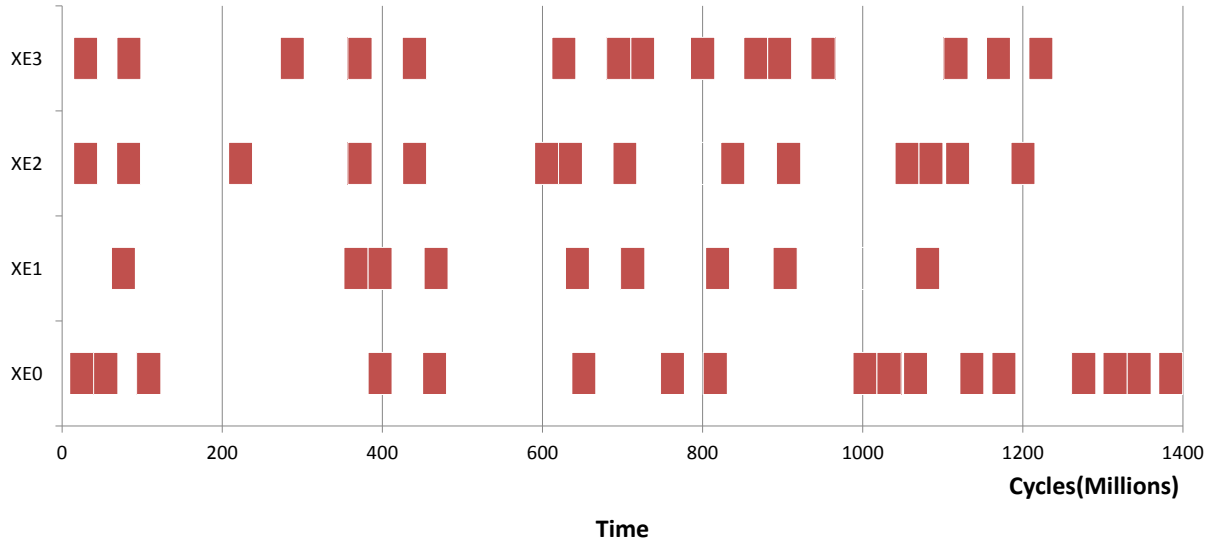

Figure 3.17: DataBlock allocation with the default policy.

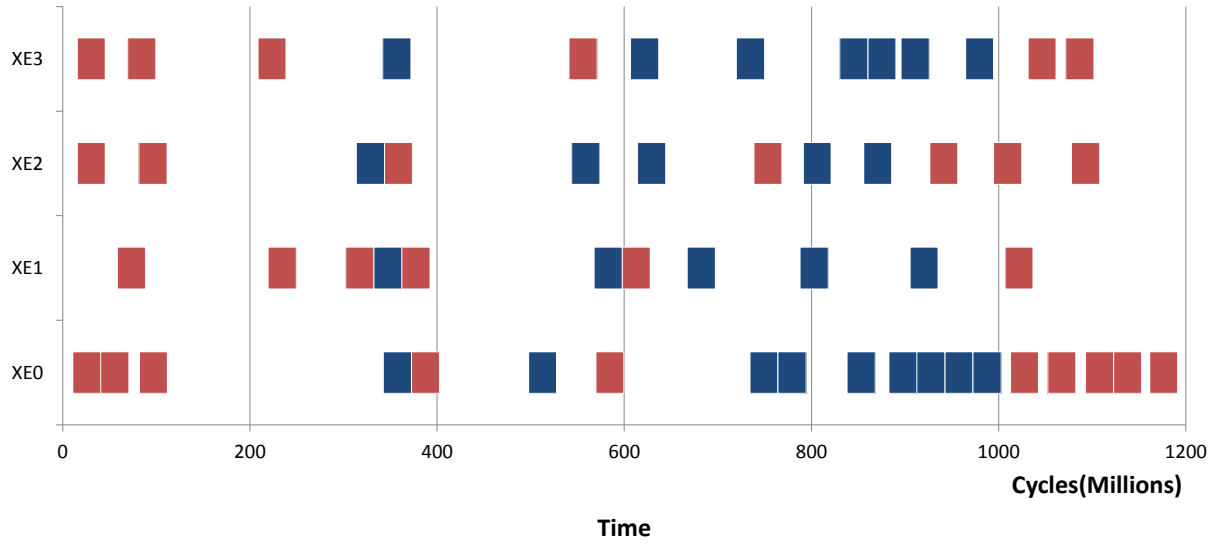

Figure 3.18: DataBlock allocation with active de-allocation policy.

With smarter management of on-chip memories, we can reduce DRAM accesses by allocating more DBs in BSM. It requires securing more capacity in BSM by moving previously-used DBs to higher level of memories (towards DRAM), or freeing previously-used DBs from BSM. The former is DB migration; the latter is DB de-allocation. DB migration is not implemented yet in OCR, thus our evaluation is for DB de-allocation.

De-allocation requires careful analysis of future reference to the DB. Otherwise, de-allocation is not complete due to future reference, or it makes loss of useful DBs. For programs which have temporary storage for the current EDT, DB de-allocation is easy. We only have to insert ocrDbDestroy at the end of the EDT. For programs which have regular one-to-one communication patterns, instrumentation is also easy because we know producer-consumer pairs easily, and the consumer can determine disposal of DBs. For example, Smith-Waterman creates temporary DBs for computation, and also creates DBs for communication. The DB for temporary use is destroyed by the EDT itself and the DBs for communication are destroyed by the consumer after use.

Cholesky has more complicated producer-consumer pairs, and it shows a 1:N producer-consumer relation. We instrument Cholesky for DB de-allocation carefully with perfect knowledge of DB use, and see the effect of DB de-allocation. Figure 3.18 shows where a new DB is allocated after de-allocation. Blue blocks show BSM-allocated DBs while red blocks still show DRAM allocation. Although more DBs are

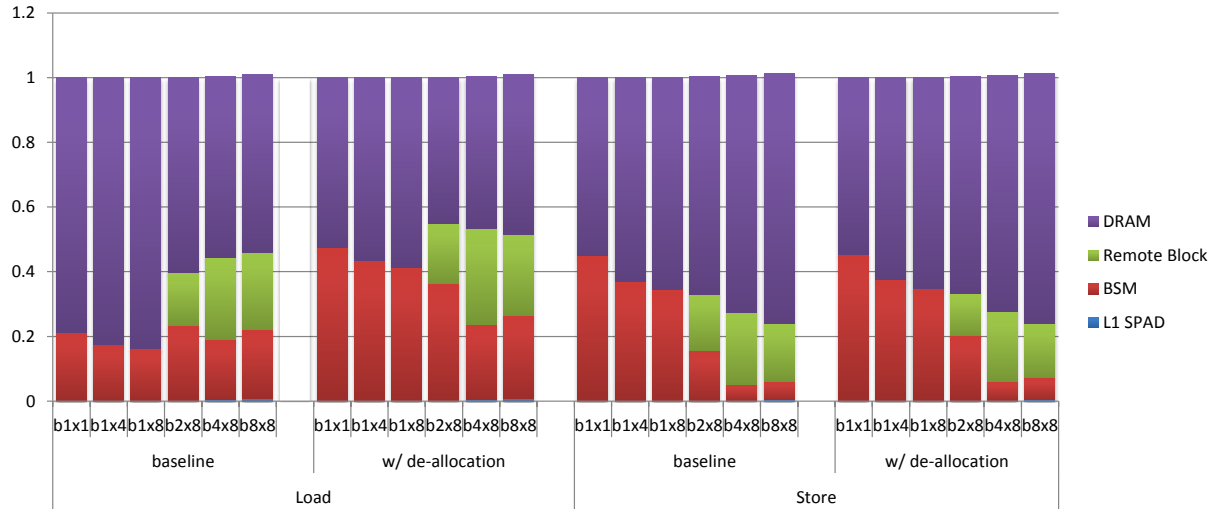

Figure 3.19: Memory access breakdown with different policies.

de-allocated from BSM, only half of newly-created DBs are allocated in BSM. This is due to OCR behavior of allocation. OCR shares the same memory allocator as the application. After the application de-allocates the DB, the freed region is also open to OCR. Another half of freed regions are reused by OCR, unexpectedly. For more intuitive results of memory de-allocation, memory allocation space for OCR needs to be separated from the memory space for the application.

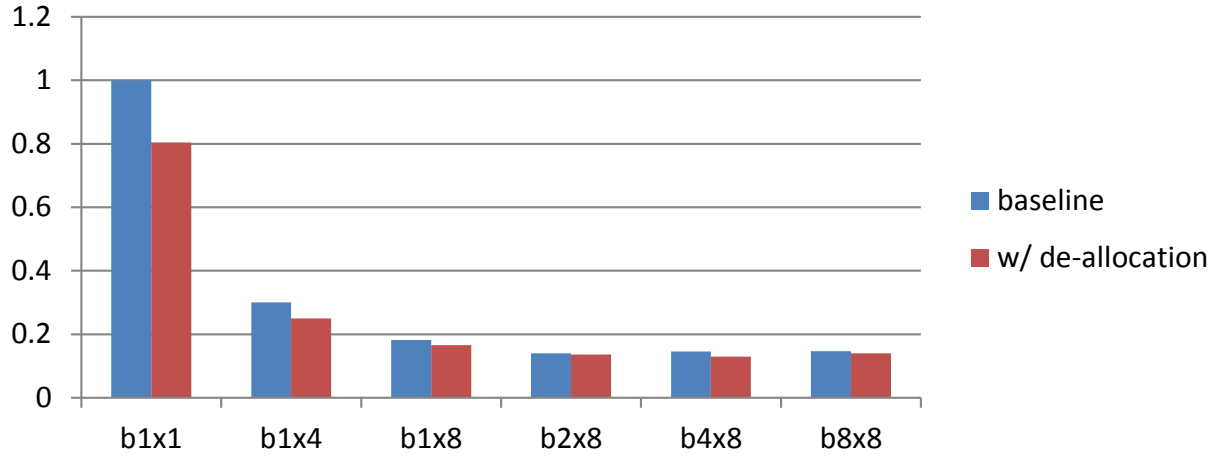

Figure 3.20: Performance improvement of data block de-allocation.

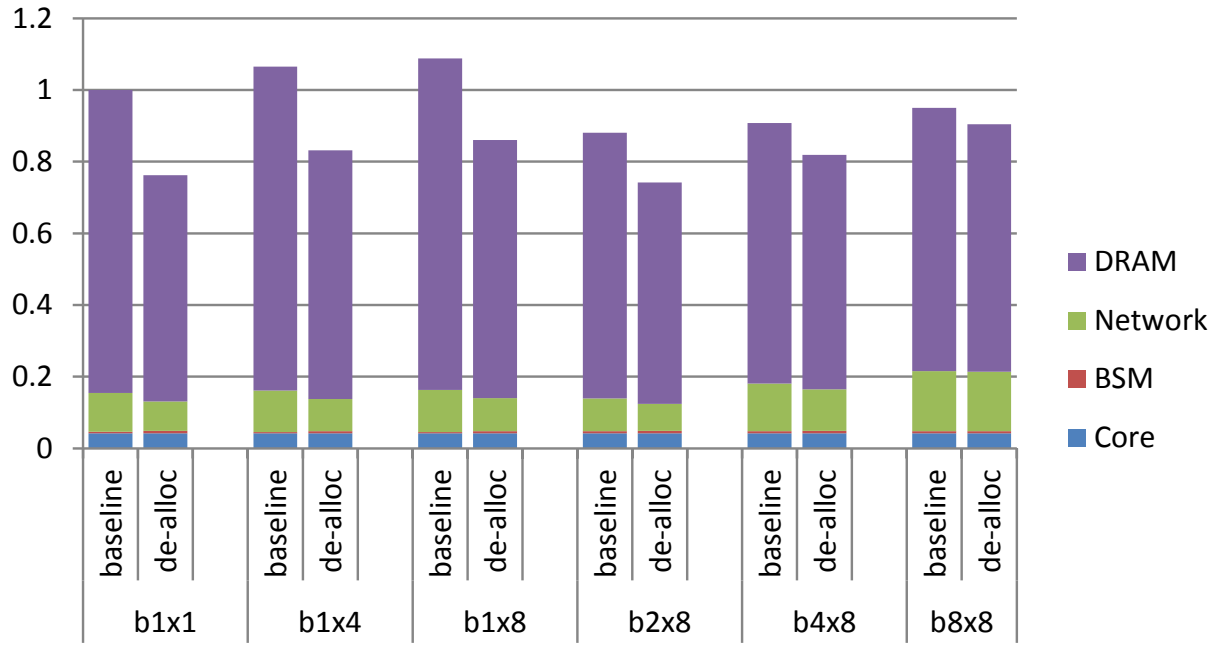

Figure 3.21: Energy improvement of data block de-allocation.

As a result of memory allocation in BSM after de-allocation, memory access breakdown in Figure 3.19 shows more accesses to BSM. The figure shows where remote memory requests are serviced from. Compared to the baseline without DB de-allocation, the new version shows more accesses to BSM, and reduces DRAM accesses. This, in turn, results in performance improvement in Figure 3.20, and energy improvement in Figure 3.21.

3.3.6 Estimation of Errors in Performance Simulation

We show performance numbers not as an absolute performance number, but as a proxy number for relative comparison. Lamport clock scheme inherently shows inflated time, which is considered upper bound of execution time (lower bound of the performance). We try to minimize errors of Lamport clock by eliminating sources of big timestamp warp. The applied methods include elimination of CE timestamp increment, no timestamp increment at busy-waiting, and no timestamp synchronization through shared memory agents.

To see how much error the simulator can have, we compare EDT execution results. While execution time is determined by many parameters, including memory access latency and CE response time, we simplify conditions to homogeneous EDT execution, mostly single source of DB location, and no complicated dependency graph. With these constraints, we compare the results of Smith-waterman with single block

execution. A very aggressive assumption about the variation of the EDT execution time is that other than the shortest execution time of the EDT is due to timing inflation of Lamport clock scheme, thus, an error of the simulator. From this assumption, the average execution time of EDTs on XEs has the error rate in Figure 3.22. The value 1.1 means maximum 10% error of Lamport clock scheme can exist compared to minimum execution time of the EDT on the XE. Average is weighted average considering different numbers of scheduled EDTs on XEs. The plot shows less-than-10% errors.

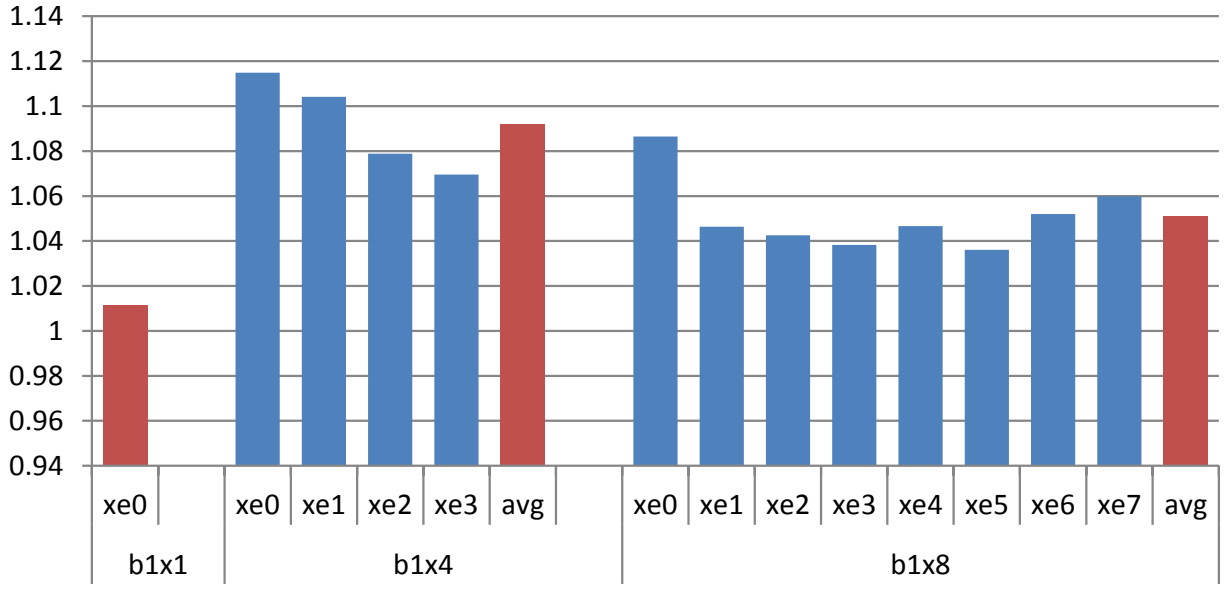

Figure 3.22: Error estimation in performance numbers.

Multi-block simulation is also similar to the previous result. But it has a hard-to-control parameter, DB location. EDT execution time varies with DB allocation places due to different memory access latencies. Multi-block configuration introduces BSM capacity, and there are variations between an EDT accessing BSM and an EDT accessing DRAM. For the EDT with the same DB location, the result is similar to the single block simulation result in Figure 3.22.

Note that our parallel speedup effect is under-estimated due to Lamport clock scheme. Adding more blocks may increase communication between blocks, leading to more timing synchronization between blocks. Since timing synchronization chooses bigger number of the timestamp, the execution time is over-estimated; parallel speedup is under-estimated.

3.4 Future Directions

Future work needs to focus on the following directions:

- First and foremost, we need to improve the efficiency of the OCR implementation on FSim. It is not possible to extract conclusions on the recommended size of EDTs in TG or the recommended frequency of DB migration in TG unless the system is reasonably tuned. We hope to see improvements over the next few months. Specifically, EDT launch and migration, and DB allocation and migration, need to be supported with low overhead. Similarly, the procedure to gather some statistics, such as the execution time, needs to be improved.

- It is also important to show that effective EDT scheduling for parallelism and DB management for locality can be done automatically.

- The FSim simulator needs to be extended to support Incoherent Caches, so that a meaningful evaluation of incoherent caches versus scratchpads can be performed in the context of TG.

Chapter 4

Related Work

Our work is based on Runnemede [14] as a baseline architecture. Runnemede has a multi-level scratchpad hierarchy with the option of implementing it with caches. Other closely related works are Rigel [33] and its successor Cohesion [34]. Rigel is a many-core accelerator architecture without hardware cache coherence. Cohesion adds support for bridging hardware cache coherence and software cache coherence. Programmers can choose the coherence scheme based on the characteristics of the application or program phase. Rigel and Cohesion with software cache coherence have similarities to ours in that they use incoherent caches, and writeback and invalidation operations to maintain coherence. We have two major improvements over these past works.

First, we define the ordering constraints of the INV and WB instructions. By doing so, we enable the use of these instructions in out-of-order cores correctly and more efficiently. Second, we provide simple yet effective approaches to program incoherent caches both within a block and across blocks. While Rigel and Cohesion have multiple levels of on-chip caches, they do not provide mechanisms for multi-level software cache coherence. INV and WB instructions work between the private L1 cache and the block-shared L2 cache. For inter-block communication, programmers need to use global memory access instructions manually. Moreover, communication occurs through the last-level global cache regardless of the current thread mapping. This inefficiency was justified with their accelerator workloads, which do not have much inter-block communication. However, exploiting block locality is required for more efficient use of memory hierarchy for general parallel programs.

In the remainder of this chapter, we summarize other related works in comparison to ours from software-only approaches to self-invalidation based hardware protocols. We also cover coarse-grained dataflow execution models as related works to the OCR model.

4.1 Software Cache Coherence

Our work in Chapter A started from reassessing early works in software cache coherence. Prior studies have developed compiler analysis techniques to generate cache coherence instructions for software managed caches. The compiler work in the appendix distinguishes itself from prior works both by being more general as well as more precise, as explained below.

Cheong et al. [16] use a data flow analysis to classify every reference to shared memory either as a memory-read or a cache-read. A read reference is marked as a memory-read if the compiler determines that the cache resident copy of the data has possibility of being stale, otherwise the reference is marked as cache-read. Invalidation is done selectively on memory-read to avoid indiscriminate invalidation of the entire cache. A limitation of the approach is that the data flow analysis is carried out at the granularity of arrays which may cause invalidations for an entire array even if two processors are accessing distinct parts of it.

Another work of Cheong et al. adapts version control scheme to coherence management. The cache maintains version number of variables and the compiler provides information about what version is valid for the use. This scheme exploits temporal locality across epochs.

Choi et al. [18] propose to improve inter-task locality in software managed caches by using additional hardware support: an epoch number is maintained at runtime using a system-wide epoch counter and each cache word is associated with a time-tag which records the epoch number in which the cache copy is created. Then they develop an epoch flow graph technique to establish conditions under which it can be guaranteed that the cached copy of a variable is not stale. This scheme improves Cheong's version control scheme by having a global epoch number, but the analysis here too treats an entire array as a single variable.

Darnell et al. [20] perform array subscript analysis to gather more accurate data dependence information and then aggregate cache coherence operations on a number of array elements to form vector operations. This method however can handle only simple array subscripts: only loop iterators are allowed as subscripts.

OBoyle et al. [39] develop techniques to identify data for Distributed Invalidation (DI) for affine loops that are run on distributed shared memory architectures. The DI scheme uses a directory to maintain coherence, and where possible it seeks to eliminate invalidation messages from the directory to remote copies and associated acknowledgments. Their analysis to minimize invalidation messages has similarities to our analysis for minimizing invalidations. But the coherence equations in DI place some restrictions on the kinds

of affine loops that can be analyzed: for example, conditional execution within the loop is disallowed, and increments of loop iterators must be unity. The approach presented in this paper efficiently handles arbitrary affine loops including those whose iterator values are lexicographically decreasing, that have a non-unit trip-count, or have a modulus operator etc., and conditionals are permitted. The DI work does not involve writebacks, which however are a part of our software cache coherence model, and we develop techniques to optimize writebacks as well. We also optimize irregular codes using an inspector-executor approach, while such codes are not optimized in the DI scheme.

Kontothanassis et al. [36] narrows the performance gap between hardware and software coherence in large systems using memory-mapped network interfaces that support a global physical address space. They showed very effective program modification techniques for performance. The four techniques used are 1) separation of synchronization variables, 2) data structure alignment and padding, 3) identification of reader-writer locks, 4) remote reference to avoid coherence management of fine grained shared data. Although some of these schemes are helpful to our target applications, we did not use the code modification suggested in their paper. Our goal is to provide hardware/software support without changing the synchronization or data layout appeared in the original codes.

Distributed shared memory (DSM) provides a shared global address space for physically distributed memory systems. TreadMarks [5] is a DSM system running on commercially available workstations and operating systems. The main focus of TreadMarks is to reduce network traffic. It uses lazy release consistency, which allows write propagation to be postponed until the next acquire. It allows multiple writers to the same page as long as different variables are written, reducing impacts of false sharing. Shasta [46] modifies the executable and adds checks for shared data availability in the local machine. If a required block is not available in the local machine, it asks the home node of the block. By emulating hardware cache coherence if required in the code, Shasta provides coherent memory image. Munin [13] supports multiple consistency protocols, which allow the programmer to choose the best protocol for the access pattern. Munin also allows multiple writers similar to TreadMarks.

DSM has similarities to ours in that it provides coherent memory images without direct hardware support for cache coherence. However, DSM usually chooses a coarse-grained granularity, such as a page, because the communication occurs between machines/clusters. Our solution provides variable granularity from a word to the entire cache because we target data sharing in many-core processor chip.

4.2 Self-Invalidation-based Hardware Coherence Protocol

Self-invalidation has been a popular approach to simplify the coherence protocol or to get software assistance. We summarize previous works and show the difference to our work.

Selective invalidation using bloom filters [7] reduces unnecessary self-invalidation. At each epoch, each core accumulates the addresses written using bloom filters. The generated signature of the written addresses is transferred with a synchronization release. A new synchronization acquirer self-invalidates its private cache using the signature received. While this work takes advantage of selective invalidation, it incurs a large overhead in lock-intensive programs. Specifically, this scheme invalidates entries in the L1 cache only *after* the lock is acquired. As a result, it increases the time duration of the critical section. This, in turn, affects all the other lock waiters. In our proposal in section 2.2.2, we provide the MEB/IEB structures to isolate critical sections from other epochs. The required invalidations and writebacks are few, and limited to addresses actually used in the critical section.

SARC coherence [32] improves directory-based coherence protocol by reducing overheads of the directory. First, it exploits tear-off read-only (TRO) copies when a cached copy is used immediately and will not be used again. TRO copies of data are self-invalidated after the use, leading to reduction of invalidation traffic. Second, it uses writer prediction to reduce indirect accesses through home directory. While SARC coherence reduces overheads of the current directory protocol, it does not reduce hardware complexity. It bases on data-race-free programs with proper synchronization, and becomes bases of VIPS protocols below.

The VIPS-M protocol [45] employs self-invalidation and self-downgrade with a delayed write-through scheme. The protocol relies on private and shared page classification. Cache lines in shared pages are self-invalidated at synchronization, and are self-downgraded using delayed write-through, which is associated with MSHR entries. Our work differs in that we use writeback L1 caches, and try to minimize unnecessary self-invalidation using MEB/IEB. If lock synchronization is used only for communication between critical sections (non for communication outside critical sections), the invalidation of the whole L1 cache at synchronization points is wasteful. Our proposals support this scenario well.

DeNovo [17] tries to simplify the hardware cache coherence protocol by using programming-model or language-level restrictions for shared memory accesses (e.g., determinism and data-race freedom). Data-race-freedom implies no concurrent writer to the cached data, and self-invalidation at next phase guarantees no stale copy is read. The benefit of DeNovo's approach is the simpler coherence protocol for design and

verification. It also reduces chip area by only registering writers instead of tracking all sharers. DeNovoND [49] extends the programming model to non-deterministic applications using lock synchronization. Access signature is made by tracking writes in the critical section, and it is used for self-invalidation by lock acquirer. DeNovoSync [48] extends its coverage to arbitrary synchronization, by registering synchronization reads. Our work seeks solutions without assumptions of the perfectly correct program. Since it is hard to guarantee complete data-race-freedom of very long legacy parallel codes, we just require programmers to identify benign data races and, if required, simply instrument them. Although we do not advocate programs with data races, we do not prohibit programs with data races.

TSO-CC [23] proposes a self-invalidation based coherence protocol for the TSO memory model. The TSO memory model guarantees write propagation in program order, and synchronization is written with that property. TSO-CC exploits the property in a lazy manner in that write propagation is guaranteed at the time synchronization variable is read rather than at the point of write. The protocol uses self-invalidation for shared lines after pre-defined number of read accesses. If newly read line is written from other cores, which means there is an update to the cache content of the local core, then self-invalidation for all shared cache lines is executed. This protocol eliminates sharer tracking and reduces write-initiated invalidation overheads. However, it introduces complicated hardware logic such as timestamp handling to reduce the range of self-invalidation.

There are two notable self-invalidation based coherence protocols for GPUs.

Temporal coherence [47] proposes a timestamp-based coherence protocol for GPU. It uses a chip-wide synchronized counter, and the timestamp is used to determine when to self-invalidate. Hardware predictor gives data with a predicted lifetime. At the predicted lifetime, the data are self-invalidated to eliminate invalidation traffic. The efficiency relies on the accuracy of the hardware predictor. Simple prediction scheme works well with highly regular characteristics of GPU kernels.

QuickRelease [29] improves the performance of release consistency on GPU. With fine-grained synchronization, release stall is a bottleneck to the performance. QuickRelease has a FIFO structure for storing written addresses to complete release operations quickly. When a release operation is enqueued to the FIFO, cache contents of written addresses registered in the FIFO are all flushed to the higher level memory. This scheme has similarities to MEB design in two aspects. First, it keeps an approximate list of written addresses. Second, the addresses in the FIFO are used for completing release operation. Major difference

from ours is that our proposal uses IEB to reduce invalidation range as well as eliminating invalidation requests. Combination of MEB/IEB separates coherence management of the critical section from other code sections and it reduces programmers' effort. QuickRelease uses invalidation broadcast. Its focus is to reduce cache look-up for the release and shorten time for the release.

Note also that, although these works show ways of implementing simpler coherence protocols using self-invalidation, they are fundamentally limited to a two-level cache hierarchy.

Concurrently with our work, Ros et al. [44] have extended self-invalidation protocols to hierarchical cluster architectures. Their work uses the hierarchical sharing status of pages to determine the level of self-invalidation. While using dynamic page-sharing information enables hierarchical coherence protocols, it complicates page management. In contrast, our work uses information extracted from the program statically, instead of dynamic information, in order to infer the target of inter-thread communication.

4.3 Coarse-grained data flow execution

The execution model of OCR is inspired by dataflow model. Dataflow model represents a program as a directed graph of data movement. The execution is driven by the availability of operands. Its static form was first proposed by Jack Dennis [22]. Static dataflow allows only one data token on the arc of graph.It was later extended to dynamic dataflow [6], which allows multiple data tokens on the arc of the dataflow and supports function calls. Dynamic dataflow exploits fine-grained parallelism; however, it suffers from overheads of associative token matching and lack of locality.

Hybrid dataflow models were introduced to reduce fine-grain synchronization costs and improve locality in dataflow model. They were implemented by combining dataflow actors into larger sequential code blocks for more locality and by having dataflow-like synchronization between threads. EARTH [50] supports an adaptive event-driven multi-threaded execution model. It has two thread levels: fibers and threaded procedures. Fibers are fine-grain threads communicating through dataflow-like synchronization. While instructions inside a fiber follows sequential execution semantics, fibers are scheduled based on data and control dependence. Threaded procedures enclose fibers, and all fibers in a threaded procedure share the local variables and input parameters. In Hybrid Technology Multi-Threaded (HTMT) program execution model [27], code is divided into many fine-grained threads. Threads are scheduled according to not only data and control dependencies but also locality requirements. HTMT extends the dataflow model with

consideration of different latencies from memory hierarchy levels.

The reincarnation of dataflow in exascale era stems from both hardware and software reasons. Using near-threshold voltage computation for energy efficiency reasons increases process variation, leading to performance variation between cores even in a chip. Software also suffers from load imbalance due to non-uniform data distribution and irregular memory accesses. These lead to a programming model with dynamic load balancing.

The execution model of OCR is runtime-managed codelets. Codelet execution model [54] consists of small sized codes, called codelets, and dependencies between codelets. A codelet is a non-pre-emptable sequence of instructions. A runtime manages dependencies between codelets and scheduling of codelets onto processing elements. Scheduling always finds available processing elements for a codelet, achieving dynamic load balancing. In OCR implementation, an EDT corresponds to a codelet, and an event corresponds to a dependency between codelets.

There are more related works to the implementation of hybrid dataflow and the codelet model.

Concurrent Collections (CnC) [12] is a high-level parallel programming model that enables programmers to write highly-parallel programs without considering a explicit degree of parallelism and scheduling onto the processing element. Programmers provide declarative annotations in existing programs, and CnC executes codes as much as parallel unless pieces of codes do not have data dependence and control dependence with each other. Mapping onto the processing element and scheduling is determined by the runtime. CnC provides a coordination language for dataflow execution, so with proper tuning specification, CnC itself is equivalent to the codelet execution model. CnC also serves as a high level language on top of OCR.

ETI SWARM [24] is one of the first implementations of the codelet execution model. It provides a low-level runtime framework that exploits dynamic task parallelism on shared memory and the distributed memory system. SWARM achieves asynchronous execution for computation and long latency communication through the use of continuation codelet. While the focus of CnC is to provide coordination capabilities to tuning experts, SWARM provides low level interface to represent dependence and scheduling constraints.

PaRSEC [10] is a framework for compiling a program to a directed acyclic graph of tasks and scheduling tasks for dynamic load balancing. PaRSEC uses job data flow (JDF) representation for dependency description. Scheduling considers JDF analysis as well as currently ready tasks. To improve data locality on NUMA architectures, scheduler also considers task locality; A newly created task is enqueud in the local

queue and a ready task in the local queue is favored for scheduling.

Chapter 5

Conclusions

Runnemede is a recently-proposed extreme-scale manycore that radically simplifies the architecture, and proposes a cluster-based on-chip memory hierarchy without hardware cache coherence. To address the challenge of programming for a memory hierarchy like Runnemede's, this paper proposed ISA support for data movement, and two simple approaches that use the ISA to program the machine. The ISA support is based on writeback and self-invalidation instructions. We showed the subtle issues that these instructions need to face, including reordering in the pipeline, and effective use of caches organized in clusters for energy efficiency. The programming approaches involved either shared-memory programming inside a cluster only, or shared-memory programming across clusters.

Our simulation results showed that the execution of applications on incoherent cache hierarchies with our support can deliver reasonable performance. For execution within a cluster, the average execution time of the applications is only 2% higher than with hardware cache coherence. For execution across multiple clusters, our applications run on average 20% faster than a naive scheme that pushes all the data to the last-level shared cache.

In our TG X-Stack project evaluation, we observed basic scalability of OCR program and runtime on Runnemede architecture. However, we also observed limitations of current architecture and simulators, and we proposed improvements in DB allocation, EDT scheduling and support for incoherent caches.

Appendix A

Compiler Support for Incoherent Caches

A.1 Overview

A computer system with software managed cache coherence does not implement cache coherence protocols in hardware. It is necessary to explicitly insert coherence instructions in a parallel program to ensure correct execution. The software orchestrates cache coherence using the following coherence primitives.

- **Writeback:** The address of a variable is specified in the instruction and if the addressed location exists in the private cache and has been modified, then it is written to a shared cache or main memory.
- **Invalidate:** The instruction causes any cached copy of the variable in the private cache to be discarded (*self-invalidation*) so that the next read to the variable fetches data from the shared cache.

The above coherence operations provide a mechanism for two processors to communicate with each other by using the shared cache: if processor A has to send an updated value of a shared variable X to processor B, then processor A issues a *writeback* instruction on X, and processor B later invalidates X so that a subsequent read to X fetches the latest value from the shared cache.

Figure A.1 shows the API for the *invalidate* and *writeback* instructions. These API functions use arguments at the granularity of word, double-word, or quad-word. The *invalidate_range* and *writeback_range* functions have a start address and number of bytes as parameters.

In this appendix chapter, we address the question of *how to automatically generate cache coherence instructions for execution on software managed caches*. The variables that potentially hold stale data — *invalidate set* — have to be identified in order that copies of those variables in the private cache are discarded; the data that are produced at a processor, but might be consumed at other processors — *writeback set* — need to be characterized so that those data are written to the shared cache and are available to other processors for future reads.

In this section, we use an example to provide an overview of the analyses performed and optimizations applied by the algorithms developed in the paper. Figure A.2 shows a 1d-jacobi stencil code. In an iteration of the stencil computation, elements of array B are updated using three neighboring elements of array A in parallel (line 4). Then, all values of array B are copied to array A in parallel (line 8). In the next iteration, the new values of array A are used to update elements of array B. This process is repeated tsteps times (loop at line 1).

The parallelism in the computation is realized using OpenMP work-sharing constructs: the loops with iterators i are declared parallel (line 3 and 7), and different iterations of those loops may be assigned to different processors to execute them in parallel. The value written to B[i] by first statement (S1) is read by the second statement (S2). Since a different processor may execute the statement producing B[i] than those that consume B[i], B[i] has to be written-back to shared cache after S1 and is invalidated before S2.

Similarly, array element A[i] is written at S2 and has uses in three iterations of the first loop at S1. The reference A[i] must therefore be written-back after S2, and invalidated before S1. The code obtained thus for Software Cache Coherence (SCC) is shown in Figure A.3.

However, there are opportunities to improve performance of the shown code on a software cache coherence system:

1. If iterations of the parallel loops in Figure A.2 are mapped to processors such that an iteration with a certain value of i is assigned to the same processor in the first and second loops (line 3 and line 7), then B[i] is produced and consumed at the same processor. Therefore, writing back of B[i] at S1, and invalidating of it at S2 can be avoided, resulting in lower overhead and better cache locality (read misses to array B in the second loop will be avoided).

   Thus, analysis to compute *invalidate* and *writeback* sets that considers the iteration-to-processor mapping can avoid many potentially conservative coherence operations.

```
invalidate_word(void *addr);
invalidate_dword(void *addr);
invalidate_qword(void *addr);
invalidate_range(void *addr, int num_bytes);

writeback_word(void *addr);
writeback_dword(void *addr);
writeback_qword(void *addr);
writeback_range(void *addr, int num_bytes);
```

Figure A.1: Coherence API list

```
1  for (t = 0; t <= tsteps-1; t++) {
2    #pragma omp parallel for
3    for (i = 2; i <= n-2; i++) {
4      S1:[i]=0.33333*(A[i-1]+A[i]+A[i+1]);
5    }
6    #pragma omp parallel for
7    for (i = 2; i <= n-2; i++) {
8      S2:A[i]=B[i];
9    }
10 }
```

Figure A.2: 1-d Jacobi stencil

```
1  for (t = 0; t <= tsteps-1; t++) {
2    #pragma omp parallel for
3    for (i = 2; i <= n-2; i++) {
4      invalidate_dword(&A[i-1]);
5      invalidate_dword(&A[i]);
6      invalidate_dword(&A[i+1]);
7      S1:B[i]=0.33333*(A[i-1]+A[i]+A[i+1]);
8      writeback_dword(&B[i]);
9    }
10   #pragma omp parallel for
11   for (i = 2; i <= n-2; i++) {
12     invalidate_dword(&B[i]);
13     S2:A[i]=B[i];
14     writeback_dword(&A[i]);
15   }
16 }
```

Figure A.3: 1-d Jacobi stencil for SCC (unoptimized)

2. Further, if iterations are mapped to processors in a block-cyclic manner, fewer invalidations and write-backs will be required. Consider for example that the iterations of the parallel loops are scheduled to processors in a block-cyclic manner with a chunk-size of 16. In that scenario, a processor writes to a consecutive set of words at S2 (line 8 in Figure A.2) — from A[k+1] to A[k+16] (for some 'k') and the same processor in the next iteration reads elements A[k] to A[k+17] at S1 (due to line 4). Therefore, only A[k], and A[k+17] have to be invalidated. Equally, other processors would only reference A[k+1] and A[k+16], and hence, only those two array cells have to be written-back.

With the above considerations, one can proceed as follows.

1) By explicitly mapping iterations of the two parallel loops to processors through the use of myid and by pinning threads to cores, we can reduce the number of coherence operations;

2) The parallel iterations are distributed among processors in a block-cyclic manner.

The resulting code is shown in Figure A.4. The very last write to a variable by any processor is also written-back so that results of the computation are available at the shared cache. The algorithms that we de-

```
1  for (t = 0; t <= tsteps-1; t++) {
2    #pragma omp parallel private(myid, i1, i2) {
3      myid = omp_get_thread_num();
4      for (i1=myid; i1<=floor((n-2)/16); i1+=8) {
5        if (t >= 1) {
6          invalidate_dword(&A[16*i1-1]);
7          invalidate_dword(&A[16*i1+16]);
8        }
9        for(i2=max(i1*16,2);i2<=min(i1*16+15,n-2);i2++) {
10         S1: B[i2]=0.33333*(A[i2-1]+A[i2]+A[i2+1]);
11       }
12       if (t == tsteps-1)
13         writeback_range(&B[i1*16], sizeof(double)*16);
14     }
15   }
16
17   #pragma omp parallel private(myid, i1, i2) {
18     myid = omp_get_thread_num();
19     for (i1=myid; i1<=floor((n-2)/16) ;i1+=8) {
20       for(i2=max(i1*16,2);i2<=min(i1*16+15,n-2);i2+=1){
21         S2: A[i2]=B[i2];
22       }
23
24       if (t <= tsteps-2) {
25         writeback_dword(&A[16*i1]);
26         writeback_dword(&A[16*i1+15]);
27       }
28       else if (t == tsteps-1) {
29         writeback_range(&A[i1*16], sizeof(double)*16);
30       }
31     }
32   }
33 }
```

Figure A.4: 1-d Jacobi stencil for execution on an 8-processor SCC system

velop in the paper automatically produce the code in Figure A.4 by performing a) an exact data dependence analysis (Section A.3.1), b) iteration-to-processor mapping aware code generation (Section A.3.2).

A.2 Background

A.2.1 Execution Model

Release Consistency: The execution of parallel programs consists of *epochs* (intervals between global synchronization points). Examples of epochs include, code executed between successive *barriers*, the code region between acquiring and releasing of a *lock*. In an epoch, data that were written potentially by other cores in previous epochs and that a core may need to read in the epoch are *invalidated*. Before the end of the epoch, all the data that a core has written in the epoch and that may be needed by other processors in

future epochs are *written-back* to the shared level cache.

Before an epoch completes, all prior memory operations, including ordinary load/store instructions and coherence instructions, are completed. Then the next epoch can start, and the following memory operations can be initiated. Further, ordering constraints between memory instructions are respected: The order of a store to address i and the following writeback for address i should be preserved in the instruction pipeline of the processor and caches. Similarly, the order of invalidation to address j and a following load from address j should be preserved in the pipeline and caches to guarantee fetching of the value from the shared cache.

Coherence Operations at Cache line granularity: Coherence operations are performed at the granularity of cache lines — all the lines that overlap with specified addresses are invalidated or written-back. If the specified data are not present in cache, then coherence instructions have no effect. In addition, *writeback instructions write back only modified words of the line*. In doing so, writeback instructions avoid the incorrectness issue that may arise from false sharing: if two processors are writing to variables that get mapped to the same cache line, and whole cache lines (and not just the *dirty* words) are written-back, then one processor's modified words may be overwritten with another processor's clean words. Therefore, per-word *dirty* bits are used to keep track of words of a cache line that are modified.

A.2.2 Notation

The code shown in Figure A.5 is used as a working example to illustrate the notation and the compiler algorithm in the next section.

Tuple sets: A tuple set s is defined as:

$$s = \{[x_1, \ldots, x_m] : c_1 \wedge \cdots \wedge c_n\}$$

where each x_i is a tuple variable and each c_j is a constraint for the range of each tuple. The iteration spaces

```
1 for (t1=0; t1<=tsteps-1; t1++) {
2   #pragma omp parallel for private(t3)
3   for (t2=0; t2<=n-1; t2++) {
4     for (t3=1; t3<=n-1; t3++) {
5       S1:B[t2][t3] = B[t2][t3+1] + 1;
6     }
7   }
8 }
```

Figure A.5: A loop nest

of statements can be represented as tuple sets. For example, the iteration space of statement S1 in the code shown in Figure A.5 can be specified as the tuple set I^{S_1}: $I^{S_1} = \{S1[t_1,t_2,t_3] : (0 \leq t_1 \leq tsteps-1) \wedge (0 \leq t_2 \leq n-1) \wedge (1 \leq t_3 \leq n-1)\}$

Access relations: An access relation r is defined as:

$$r = \{[x_1,\ldots,x_m] \mapsto [y_1,\ldots,y_n] : c_1 \wedge \cdots \wedge c_p\}$$

where each x_i is an input tuple variable, each y_j is an output tuple variable and each c_k is a constraint. Array accesses appearing in the code may be modeled as access relations from iteration spaces to access functions of the array references. The two accesses to array 'B' in Figure A.5, B[t2][t3] and B[t2][t3+1], are represented as the following relations:

$$r^{S_1}_{write} = \{S1[t_1,t_2,t_3] \mapsto B[t'_2,t'_3] : (t'_2 = t_2) \wedge (t'_3 = t_3)\}$$
$$r^{S_1}_{read} = \{S1[t_1,t_2,t_3] \mapsto B[t'_2,t'_3] : (t'_2 = t_2) \wedge (t'_3 = t_3+1)\}$$

The Apply Operation: The apply operation on a relation r and a set s produces a set s' denoted by, $s' = r(s)$ and is mathematically defined as:

$$(\vec{x} \in s') \Longleftrightarrow (\exists \vec{y} \text{ s.t. } \vec{y} \in s \wedge (\vec{y} \mapsto \vec{x}) \in r)$$

The set of array elements accessed by an array reference in a loop (data-footprint) may be derived by *applying* access function *relations* on the iteration space *sets*. For the array accesses in the example code shown in Figure A.5, data-footprints of the two accesses are: $r^{S_1}_{write}(I^{S_1}), r^{S_1}_{read}(I^{S_1})$.

The Inverse Operation: The inverse operation $r = r_k^{-1}$ operates on an access relation r_k to produce a new access relation r such that r has the same constraints as r_k but with the input and output tuple variables swapped. $(\vec{x} \mapsto \vec{y} \in r) \Longleftrightarrow (\vec{y} \mapsto \vec{x} \in r_k)$.

A.2.3 Polyhedral Dependences

In the Polyhedral model, for affine array index, dependence analysis [25] can precisely compute flow (Read After Write — RAW) and output (Write After Write — WAW) dependences between dynamic instances of

statements. The dependences are expressed as maps from source iterations to target iterations involved in the dependence.

The flow dependence determined by polyhedral dependence analysis (for example, using ISL [51]) for the code in Figure A.5 is:

$$\mathcal{D}_{flow} = \{S1[t_1,t_2,t_3] \mapsto S1[t_1+1,t_2,t_3-1] : (0 \leq t_1 \leq tsteps-2) \wedge (0 \leq t_2 \leq n-1) \wedge (2 \leq t_3 \leq n-1)\}$$

The relation characterizes the flow dependence that exists between the write reference B[t2][t3] and the read reference B[t2][t3+1]. An analysis tool like ISL can also be used to emit information regarding *live-in* data: data that are read in the loop but are not produced by any statement instances in the scope of analysis. A list containing maps from an iteration point that reads live-in data to the live-in array elements that it reads is computed. For the running example, the live-in maps are:

$$\mathcal{D}_{live-in} = \{S1[0,t_2,t_3] \mapsto B[t_2,t_3+1] : 0 \leq t_2 \leq n-1 \wedge 1 \leq t_3 \leq n-2;$$
$$S1[t_1,t_2,n-1] \mapsto B[t_2,n] : 0 \leq t_1 \leq tsteps-1 \wedge 0 \leq t_2 \leq n-1\}$$

The two maps capture live-in data read for read reference B[t2][t3+1].

A.3 Compiler Optimization for Regular Codes

The iteration space of an epoch in a parallel loop is modeled by considering iterator values of the parallel loop and its surrounding loops as parameters. In the parallel loop in Fig. A.5, the t2 loop is parallel and an iteration of t2 constitutes a *parallel task* executed in an epoch. Its iteration space is modeled by considering values of iterators t1 and t2 as parameters — t_p and t_q respectively:
$I^{S_1}_{current} = \{S1[t_1,t_2,t_3] : (t_1 = t_p) \wedge (t_2 = t_q) \wedge (1 \leq t_3 \leq n-1)\}$.

A.3.1 Computation of Invalidate and Writeback Sets

Invalidate Set: Algorithm 1 shows how the invalidate set for a parallel task is computed. It is computed by forming the union of invalidate data sets corresponding to all statements within the parallel loop by iterating over each statement. For each statement S_i, first the source iterations of the dependence — I_{source}

— whose target iterations are in the current slice for that statement — $I^{Si}_{current}$ — are determined by applying the inverse relation of the flow dependence. From this set, any of the source iterations that lie in the current slice — $\bigcup_{Sj \in stmts} I^{Sj}_{current}$, are removed from I_{source} because the source and target iterations are run on the same processor and no coherence instruction is needed. The array elements written by iterations of I_{source} are placed in the set of data elements for which invalidation coherence instructions must be issued to guarantee coherence. To this set is added the live-in list corresponding to data elements that come in live from outside the analyzed region.

Require: Flow Dependences : $\mathcal{D}_{flow}$, Live-in read maps : $\mathcal{D}_{live\_in}$, Current Iteration Slices: $I_{current}$, Write maps: r_{write}
Ensure: Statement and Invalidate set pairs: $D^{S_i}_{invalidate}$

1: **for all** statements - Si **do**
2: $\quad D^{S_i}_{invalidate} \leftarrow \phi$
3: $\quad I_{source} \leftarrow \mathcal{D}^{-1}_{flow}(I^{Si}_{current}) \setminus (\bigcup_{Sj \in stmts} I^{Sj}_{current})$
4: $\quad D_{inflow} \leftarrow \bigcup_{Sj \in stmts} r^{Sj}_{write}(I_{source})$
5: $\quad D_{live\_in\_data} \leftarrow \mathcal{D}_{live\_in}(I^{Si}_{current})$
6: $\quad D^{S_i}_{invalidate} \leftarrow (D_{inflow} \cup D_{live\_in\_data})$
7: **end for**

Algorithm 1: Compute Invalidate Set

Example: The application of the algorithm to the running example results in the following invalidate set: $D^{S_1}_{invalidate} = \{[t_q, i_1] : 2 \leq i_1 \leq n\}$. The array elements read in the parallel task are marked for invalidation.

Writeback Set: Algorithm 2 shows how we compute the writeback set for a parallel task that possibly has multiple statements in it. To find the writeback set corresponding to a statement S_i, first all target iterations (I_{target}) of all dependences are identified whose source iterations lie in $I^{Si}_{current}$. Those target iterations that are within the same parallel task — $\bigcup_{Sj \in stmts} I^{Sj}_{current}$ are removed from I_{target} (line 3). Then the inverse dataflow relation is applied to this set and the intersection to the current iteration slice is computed (line 4) to identify the source iterations ($I_{producer}$) in the slice that write values needed outside this slice. These values must be part of the writeback set. Further, if a write by an iteration is the last write to a certain variable, it must also be written back since it represents a live-out value from the loop. The iterations that are not sources of any output dependencies produce live-out values. Such iterations are determined by forming the set difference between $I^{Si}_{current}$ and domain of output dependences — $dom\ \mathcal{D}_{output}$.

Example: The algorithm produces the following writeback set for the example in Figure A.5: $D^{S_1}_{writeback} =$

Require: Flow Dependences : $\mathcal{D}_{flow}$, Output Dependences : $\mathcal{D}_{output}$, Current Iteration Slices: $I_{current}$, Write maps: r_{write}
Ensure: Statement and Writeback set pairs: $D^{S_i}_{writeback}$

1: **for all** statements - Si **do**
2: $\quad D^{S_i}_{writeback} \leftarrow \phi$
3: $\quad I_{target} \leftarrow \mathcal{D}_{flow}(I^{Si}_{current}) \setminus (\bigcup_{Sj \in stmts} I^{Sj}_{current})$
4: $\quad I_{producer} \leftarrow \mathcal{D}^{-1}_{flow}(I_{target}) \cap I^{Si}_{current}$
5: $\quad D_{outflow} \leftarrow r^{Si}_{write}(I_{producer})$
6: $\quad I_{live\_out} \leftarrow I^{Si}_{current} \setminus dom\ \mathcal{D}_{output}$
7: $\quad D_{live\_out\_data} \leftarrow r^{Si}_{write}(I_{live\_out})$
8: $\quad D^{S_i}_{writeback} \leftarrow (D_{outflow} \cup D_{live\_out\_data})$
9: **end for**

Algorithm 2: Compute Writeback Set

$\{[t_q, i_1] : (t_p \leq tsteps - 2 \wedge 2 \leq i_1 \leq n-1) \vee (t_p = tsteps - 1 \wedge 1 \leq i_1 \leq n-1)\}$.

For 0 to tsteps-2 iterations of the outermost t1 loop, only elements B[t2][2:n-1] need to be written back as they will be read in the next iteration of t1 loop. Array cell B[t2][1] does not need to be written back because it is overwritten in a later t1 iteration and its value is not read. But the very last write to B[t2][1], i.e., when t1 = tsteps-1, has to be written back as it is a live-out value of the loop.

Code Generation The invalidate and writeback sets are translated to corresponding cache coherence instructions by generating a loop to traverse elements of the sets using a polyhedral code generator — ISL [51]. The invalidations and writebacks are combined into coherence range functions whenever elements of a set are contiguous in memory: when the inner-most dimension of the array is the fastest varying dimension of the loop.

A.3.2 Optimization: Analysis Cognizant of Iteration to Processor Mapping

The techniques described until now do not assume any particular mapping of iterations to processors. However, if a mapping of processors to iterations is known, many invalidations and write-backs could possibly be avoided. For example, in the code shown in Figure A.5, the flow dependence (mentioned in §A.2.3) is: $S1[t_1, t_2, t_3] \mapsto S1[t_1 + 1, t_2, t_3 - 1]$. If parallel iterations of the 't2' loop are mapped to processors such that an iteration with a particular 't2' value always gets mapped to the same processor, the source and target iterations of the flow dependence get executed on the same processor, making invalidations and write-backs

```
1  for (t1=0; t1<=tsteps-1; t1++) {
2    #pragma omp parallel private (myid,t2,t3) {
3      myid = omp_get_thread_num();
4      for (t2=myid; t2<=n-1; t2+=8) {
5        if (t1 == 0) {
6          invalidate_range(&B[t2][2], sizeof(double)*(n-2));
7        }
8        invalidate_dword(&B[t2][n]);
9        for (t3=1;t3<=n-1;t3++) {
10         S1: B[t2][t3] = B[t2][t3+1] + 1;
11       }
12       if (t1 == tsteps-1) {
13         writeback_range(&B[t2][1], sizeof(double)*(n-1));
14       }
15     }
16   }
17 }
```

Figure A.6: Optimized loop nest for SCC

due to the dependence unnecessary.

In order to incorporate this optimization, Algorithm 1 and 2 are modified to take iteration to processor mapping into account. Line 3 of Algorithm 1 is now changed to:

$I_{source} \leftarrow \mathcal{D}^{-1}_{flow}(I^{Si}_{current}) \setminus (\bigcup_{Sj \in stmts} I^{Sj}_{current} \cup I_{same\_proc})$

and line 3 of Algorithm 2 is changed to:

$I_{target} \leftarrow \mathcal{D}_{flow}(I^{Si}_{current}) \setminus (\bigcup_{Sj \in stmts} I^{Sj}_{current} \cup I_{same\_proc})$,

where I_{same_proc} is the set of iterations that is executed on the same processor as the processor on which $I_{current}$ is executed.

For the working example, let us say that the OpenMP scheduling clauses specify that iterations are cyclically mapped onto processors and the number of processors used is 8. Then, we encode that information into the following iteration to processor map: $r_{i2p} = \{S1[t_1,t_2,t_3] \mapsto [t'_2] : t'_2 = t_2 \mod 8\}$. The parallel region code is all the iterations that are mapped to a parametric processor 'myid': $I_{my\_proc} = r^{-1}_{i2p}(myid)$. The iteration set $I^{S_1}_{current}$ is a subset of I_{my_proc} with the values of the t1 and t2 loop iterators parameterized. Using the modified algorithms, the cache coherence code generated for the working example is presented in Figure A.6. In the optimized code, only the live-in data is invalidated: elements B[t2][2 to n] at time-step $t_1 = 0$, only a single element — B[t2][n] at later time-steps, since other elements are written to by the same processor ensuring that the updated values are present in the processor's private cache. Only the live-out data is written back at the last time-step: $t_1 = tsteps - 1$.

A.4 Compiler Optimization for Irregular Codes

The irregular computations – codes whose data flow is not expressed in an affine index form and control flow may not be determined precisely, are handled with a combination of compile-time and run-time techniques. We first describe a completely general scheme (Section A.4.1) that preserves data locality in caches within an *epoch* but not across *epochs*. Then, we present specialized methods (Section A.4.2, Section A.4.3) for certain classes of irregular codes which preserve data locality across epoch boundaries.

A.4.1 Bulk Coherence Operations: Basic Approach

The tasks that are executed in an epoch (interval between synchronization points) by construction do not have any dependences between them (otherwise, the dependences would induce serialization of tasks and hence, the tasks would have to be executed in different epochs). Therefore, all data accessed *within* an epoch can be safely cached and cache coherence is not violated.

```
invalidate_all();
writeback_all();
```

Figure A.7: Coherence API for conservative handling

For irregular applications that have non-affine references and hence, are not amenable to the analysis presented in the previous section, software cache coherence is achieved conservatively: at the beginning of an epoch, the entire private cache is invalidated and at the end of the epoch, all data that are written in the epoch (dirty words) are written to the shared cache. The coherence API functions shown in Figure A.7 are inserted in the parallel program at epoch boundaries to conservatively manage software coherence. The basic approach outlined above preserves intra-epoch cache data locality, but cannot exploit any temporal locality that exists across epoch boundaries.

A.4.2 Inspector-Executors

Many scientific applications use sparse and irregular computations and are often iterative in nature and furthermore, the data access pattern remains the same across iterations. Examples include programs for solving partial differential equations, irregular stencils, the conjugate gradient method for solving systems of linear equations which uses sparse matrix-vector multiplications, atmospheric simulations that use semi-regular grids.

```
1  while (converged == false) {
2    #pragma omp parallel for
3    for (i=0; i<n; i++) {
4      read A[B[i]]; /*data-dependent access*/
5    }
6
7    #pragma omp parallel for
8    for (i=0; i<n; i++) {
9      write A[C[i]]; /*data-dependent access*/
10   }
11   /*Setting of converged variable not shown*/
12 }
```

Figure A.8: A time-iterated loop with irregular data references

For such codes, we propose the use of *inspectors* [21, 53] to gather information on irregular data accesses so that coherence operations are applied only where necessary. The inspectors that are inserted in the parallel codes are themselves parallel and are lock-free. The cost of inspectors is amortized by the ensuing selective invalidations of data and thus fewer unnecessary L1 cache misses over many iterations of the iterative computation. Figure A.8 shows an iterative code that has data-dependent references to a one-dimensional array, viz., A[B[i]] and A[C[i]]. We first illustrate the inspector approach for the simple example. The ideas are more generally applicable in the presence of multiple arrays and multi-dimensional arrays.

The inspector-code determines if a) the write performed at a thread has readers at other threads: if that is the case, the variable has to be written-back to the shared cache so that other threads will be able to obtain the updated value of the variable. b) the variable being read at a thread was written by another thread: if yes, the variable has to be invalidated at the private cache so that the fresh value is got from the shared cache. Figure A.9 presents the inspector-inserted parallel code corresponding to the iterative code shown in Figure A.8 for execution on software managed caches.

Two shadow arrays — A_thread and A_conflict for array A that has data-dependent accesses are initialized (lines 4, 5). In the *first phase*, A_thread records the ids of the threads that write to array cells (line 12). In the *second phase*, if an array cell is read by a thread different from the writer thread, the corresponding cell in A_conflict array is set to 1 (line 22). Since the computation loops are parallel, the inspection is also carried out in parallel. Consequently, accesses to arrays A_thread and A_conflict are guarded with coherence instructions. If there are multiple readers for an array cell then more than one thread may set the respective cell of A_conflict to 1 in phase two and multiple threads will write-back the same value, namely 1 to the shared cache (in line 23). Since the same value is being written, any ordering of writes by different threads

```
1  /*Inspector code begins*/
2  #pragma omp parallel for
3  for(i=0; i<n; i++) {
4    A_thread[i] = -1;
5    A_conflict[i] = 0;
6    writeback_word(&A_thread[i]);
7    writeback_word(&A_conflict[i]);
8  }
9  //Phase 1: Record writer thread ids
10 #pragma omp parallel for
11 for(i=0; i<n; i++) {
12   A_thread[C[i]] = myid;
13   writeback_word(&A_thread[C[i]]);
14 }
15 //Phase 2: Mark conflicted if writer and reader threads are not the same
16 #pragma omp parallel for
17 for(i=0; i<n; i++) {
18   invalidate_word(&A_thread[B[i]]);
19   if(A_thread[B[i]] != -1 && A_thread[B[i]] != myid) {
20     A_conflict[B[i]] = 1;
21     writeback_word(&A_conflict[B[i]]);
22   }
23 } /*Inspector code ends*/
24 #pragma omp parallel { invalidate_all(); }
25 while (converged == false) {
26   #pragma omp parallel for
27   for (i=0; i<n; i++) {
28     if (A_thread[B[i]] != -1 && A_thread[B[i]] != myid)
29       invalidate_word(&A[B[i]]);
30     read A[B[i]];
31   }
32   #pragma omp parallel for
33   for(i=0; i<n; i++) {
34     write A[C[i]];
35     if (A_conflict[C[i]] == 1)
36       writeback_word(&A[C[i]]);
37   }
38   /*Setting of converged variable not shown*/
39 }
40 #pragma omp parallel { writeback_all(); }
```

Figure A.9: An iterative code with irregular data references for SCC system

works.

Later in the computation loops, a thread invalidates a variable (line 36) before reading it if the variable has a writer (as opposed to read-only data) and that writer is a different thread. A thread after writing to a variable, writes it back (line 44) if the variable is marked conflicted.

A.4.3 Exclusion of Read-Only Data from Coherence

For irregular codes whose data access patterns potentially change with each iteration, we adopt a conservative approach that yet excludes read-only data from coherence enforcement and thus, is more accurate than

```
1  /*Prologue begins*/
2  writeback_all();
3  #pragma omp parallel
4  { invalidate_all(); }
5  /*Prologue ends*/
6
7  while (condition){
8    #pragma omp parallel
9    {
10     /*regular/irregular code*/
11   }
12 }
13
14 /*Epilogue begins*/
15 #pragma omp parallel
16 { writeback_all(); }
17 invalidate_all();
18 /*Epilogue ends*/
```

Figure A.10: A loop with bulk coherence operations at parallel region boundaries

a full invalidation and writeback approach outlined earlier. We consider *parallel regions* — parallel loops along with surrounding looping structures and perform analysis of the parallel region as a stand-alone unit. The *read-only data of the parallel region* need not be invalidated/written-back. Only those variables that are both written and read in the parallel region are invalidated and written-back at epoch boundaries.

For this scheme to work, however, the following conditions have to be met:

1. None of the processors should have cached stale values of read-only data of the parallel region. (This could happen, for example, when a program has a parallel region $\mathcal{P}$ followed by a sequential segment $\mathcal{Q}$ and later a parallel region $\mathcal{R}$. And, variable x is read-only in $\mathcal{P}$ and $\mathcal{R}$, but is modified in $\mathcal{Q}$).

2. Since, in the parallel region coherence is enforced only on data that are both read and written, for *written-but-not-read* data coherence operations should be introduced following the parallel region to ensure that future accesses to them get updated values.

To meet condition 1), a *prologue* is introduced that writes back all dirty words from the master thread and then does a full invalidation of caches at all threads. Condition 2) is fulfilled by writing-back all dirty words from all threads and doing a full-invalidation by the master thread in an *epilogue*. The code shown in Figure A.10 uses the outlined approach. Algorithm 3 presents the overall parallel-region analysis technique.

Overall Approach The overall compiler analysis operates by checking if it can apply optimizations in the following order (most constrained to unconstrained): 1) regular programs with static schedules (§A.3.2),

2) regular programs with dynamic schedules (§A.3.1), 3) inspector-executor (§A.4.2): if the array-indexing variables (e.g., in reference A[B[i]], we refer to B[i] as the indexing variable) are not written inside the time-loop and the control flow, if any, does not contain variables that are written inside the time-loop. 4) exclusion of read-only data (§A.4.3), finally, for the rest, 5) bulk coherence operations (§A.4.1).

Table A.1: Benchmarks. Legend: #PL: Number of Parallel Loops; #PLI: Number of Parallel Loops containing irregular accesses

Benchmark	Description	Does application have irregular accesses?	#PL	#PLI	Techniques used
gemm	Matrix-multiply : $C = \alpha.A.B + \beta.C$	No	2	0	Polyhedral
gemver	Vector Multiplication and Matrix Addition	No	3	0	Polyhedral
jacobi-1d	1-D Jacobi stencil computation	No	2	0	Polyhedral
jacobi-2d	2-D Jacobi stencil computation	No	2	0	Polyhedral
LU	LU decomposition	No	1	0	Polyhedral
trisolv	Triangular solver	No	1	0	Polyhedral
CG	Conjugate Gradient method	Yes	3	1	Inspector-Executor + Polyhedral
backprop	Pattern recognition using unstructured grid	Yes	2	0	Bulk + Polyhedral
hotspot	Thermal simulation using structured grid	Yes	2	0	Bulk + Polyhedral
kmeans	Clustering algorithm used in data-mining	Yes	1	1	Exclusion of RO data
pathfinder	Dynamic Programming for grid traversal	Yes	1	1	Exclusion of RO data
srad	Image Processing using structured grid	Yes	2	2	Inspector-Executor

Require: AST of Parallel region: $\mathcal{P}$
Ensure: AST of Parallel region for SCC: $\mathcal{P}_{SCC}$
1: $Prologue \leftarrow$ API to write-back all dirty words from master thread; API to invalidate entire cache of all threads
2: $Read\_Set \leftarrow$ Arrays and scalars that are read in $\mathcal{P}$
3: $Write\_Set \leftarrow$ Arrays and scalars that are written in $\mathcal{P}$
4: $Coherence\_Set \leftarrow Read\_Set \cap Write\_Set$
5: **for all** epoch code $e \in \mathcal{P}$ **do**
6: $\quad Invalidate\_Set_e \leftarrow Read\_Set_e \cap Coherence\_Set$
7: $\quad Writeback\_Set_e \leftarrow Write\_Set_e \cap Coherence\_Set$
8: $\quad$ Insert API for $Invalidate\_Set_e$ and $Writeback\_Set_e$
9: **end for**
10: $Epilogue \leftarrow$ API to write-back all dirty words from all threads; API to invalidate entire cache of master thread
11: $\mathcal{P}_{SCC} \leftarrow$ Append $\{Prologue, \mathcal{P}, Epilogue\}$

Algorithm 3: Generate Coherence Instructions using Parallel Region Analysis

Table A.2: Simulator parameters

Processor chip	8-core multicore chip
Issue width; ROB size	4-issue; 176 entries
Private L1 cache	32KB Write-back, 4-way, 2 cycle hit latency
Shared L2 cache	1MB Write-back, 8-way, multi-banked 11 cycle round-trip time
Cache line size	32 bytes
Cache coherence protocol	Snooping-based MESI protocol
Main Memory	300 cycle round-trip time

A.5 Experimental Evaluation

We evaluate the performance of compiler-generated coherence instructions for execution of parallel programs on software managed caches. The main goal of the compiler support developed in the paper is to insert coherence instructions — *invalidate* and *writeback* functions only where necessary. The conservative invalidations (of non-stale data) result in read misses which lead to degraded performance relative to a hardware coherence scheme. Therefore, to assess efficacy of the compiler techniques, we compare *read misses in L1 caches*, and *execution time* on software and hardware managed caches. (The number of misses at the shared cache is unaffected and will be the same for software and hardware cache coherence.)

Conservative coherence operations in software scheme increase accesses to the shared cache and also, cause increased traffic on the system bus. The hardware cache coherence protocol uses control messages to maintain coherence, which a software scheme does not. Therefore, if the software coherence mechanism results in comparable cache misses as a hardware protocol, then the software coherence also reduces network traffic and cache energy. We therefore measure the *number of words transferred on the system bus* and *cache energy* by software and hardware coherence systems.

A.5.1 Benchmarks

The benchmark programs used for the experiments and their characteristics are listed in Table A.1. The benchmark programs — gemm, gemver, jacobi-1d, jacobi-2d, LU, trisolv are taken from PolyBench benchmark suite [42]. The PolyBench benchmark suite is a collection of widely used linear algebra, and stencil codes. The codes are parallelized using a polyhedral compiler – PoCC [41]. All array references in the Poly-

Bench programs are affine and coherence instructions are generated using Polyhedral techniques presented in Section A.3.

The backprop, hotspot, kmeans, pathfinder, srad applications are taken from Rodinia suite [15], and they contain affine as well as irregular data references. The Rodinia suite provides parallel programs from various application domains. At the parallel region boundaries in Rodinia applications, bulk coherence instructions (invalidate_all and writeback_all) are applied (Section A.4.1) while the parallel regions are optimized. Inspector-executor method (Section A.4.2) is used for irregular data references in CG and srad applications. Exclusion of read-only data optimization described in Section A.4.3 is employed in kmeans and pathfinder codes. The parallel loops in backprop and hotspot benchmarks are amenable to polyhedral analysis and therefore, bulk coherence operations are inserted at the beginning and end of parallel regions, and coherence operations in parallel loops are derived using polyhedral algorithms (Section A.3).

A.5.2 Set-up

The snooping-bus MESI protocol hardware coherence (referred to as *HCC* in the following text), and software cache coherence (referred to as *SCC*) have been implemented in an architectural multi-processor simulator — SESC [40]. Details of the simulator setup are described in Table A.2.

We compare performance and energy of the following four coherence schemes:

1. **HCC:** Parallel programs are executed using MESI hardware coherence.

2. **SCC-basic:** The coherence instructions are inserted without iteration-to-processor aware analysis for affine references and without the use of inspector-executor or read-only data exclusion scheme for irregular accesses. That is, coherence instructions are generated with methods described in Sections A.3.1 and A.4.1 only without further optimizations. The resulting codes are run on software managed caches.

3. **SCC-opt:** The coherence management is optimized using compiler optimizations presented, and the resulting programs are executed on software managed caches.

4. **HCC-opt:** To study if any optimizations applied to SCC codes (such as explicit mapping of iterations to processors) can also benefit the benchmarks for hardware coherence, SCC-opt programs are adapted

to run on HCC systems: coherence operations and any *inspectors* inserted are removed from SCC-opt codes and these variants are run on the HCC system.

The performances of only parallel parts of benchmarks are measured — sequential initialization and finalization codes are excluded from measurements because the performance of sequential code is expected to be the same on SCC and HCC systems. Threads are pinned to cores for both schemes.

A.5.3 Performance Results

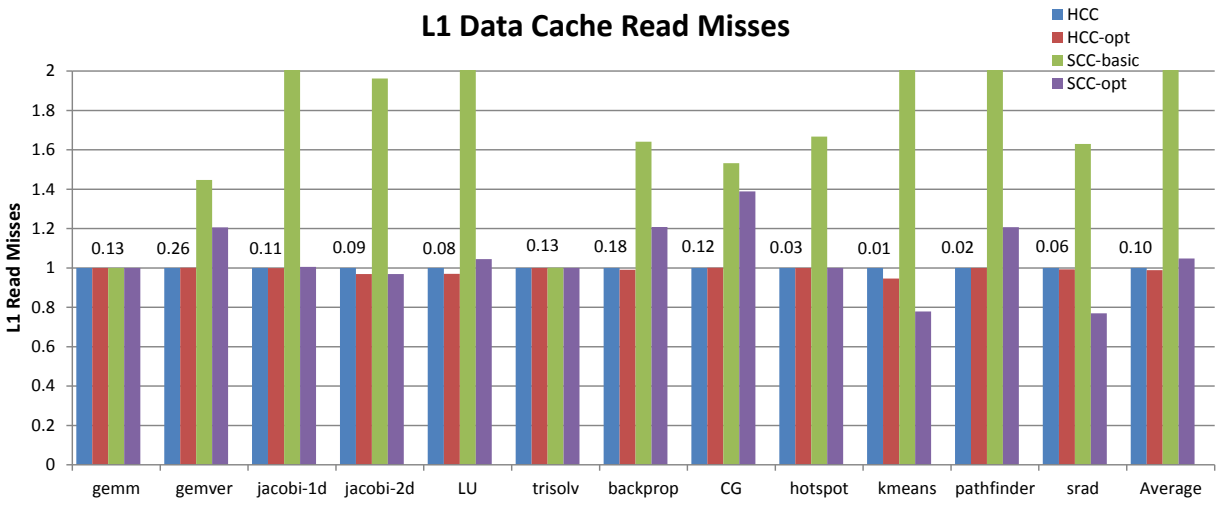

Figure A.11: L1 data cache read misses (lower, the better). The L1 read miss ratios for HCC are also shown as numbers above the bar.

Figure A.11 plots the read misses in L1 cache. The number of L1 read misses and execution cycles are normalized with respect to HCC statistics (the number of misses and execution cycles of HCC is considered 1). The L1 read miss ratios (fraction of L1 reads that are misses) for HCC are also indicated in the graph. On average across benchmarks, HCC-opt has the same number of cache misses as HCC; SCC-basic suffers 135% more misses and SCC-opt experiences only a 5% increase (avg. column in the graph). The average of normalized execution time for the three variants — HCC-opt, SCC-basic, and SCC-opt are, 0.98, 1.66, and 0.97 respectively. We observe that SCC-opt greatly improves performance over SCC-basic and brings down cache misses comparable to those of HCC. Further, performance of HCC-opt is very similar to that of HCC.

The gemm and trisolv benchmarks exhibit the so-called *communication free* parallelism: the outer loop in these codes is parallel. Therefore, there is no communication between processors induced by data depen-

dences. All code variants of gemm and trisolv have virtually the same number of cache misses and execution cycles. In applications that have irregular references, namely backprop, CG, hotspot, kmeans, pathfinder, srad, the parallel region boundaries are guarded with full-invalidation and full-writeback instructions (described in A.4.3) The affine accesses in the parallel regions are optimized; irregular accesses are handled using *inspectors* or invalidation and write-back of entire arrays that are both written and read in the parallel region (read-only arrays and scalars are excluded). For backprop and pathfinder, full invalidation of cache at parallel region boundaries results in some loss of data locality which results in increased L1 cache read misses. The CG and srad benchmarks have iterative loops and irregular accesses whose indexing structures do not change across iterations. Therefore, for those two benchmarks, *inspector* codes are inserted for deriving coherence operations. The inspectors contribute to a certain number of L1 read misses. The reduced cache misses in SCC-opt of srad compared to its HCC counterpart is an artifact of the interaction that exists between cache coherence and cache replacement policy (LRU): false-sharing in HCC can cause soon-to-be-reused data to be evicted, which favors SCC. The migratory writes may sometimes cause invalidations of not-to-be-reused data and thus, making way for other to-be-reused data and this benefits HCC. Conversely, the gemver is an example of hardware cache coherence working to HCC advantage, where the number of misses for HCC is lower compared to SCC-opt.

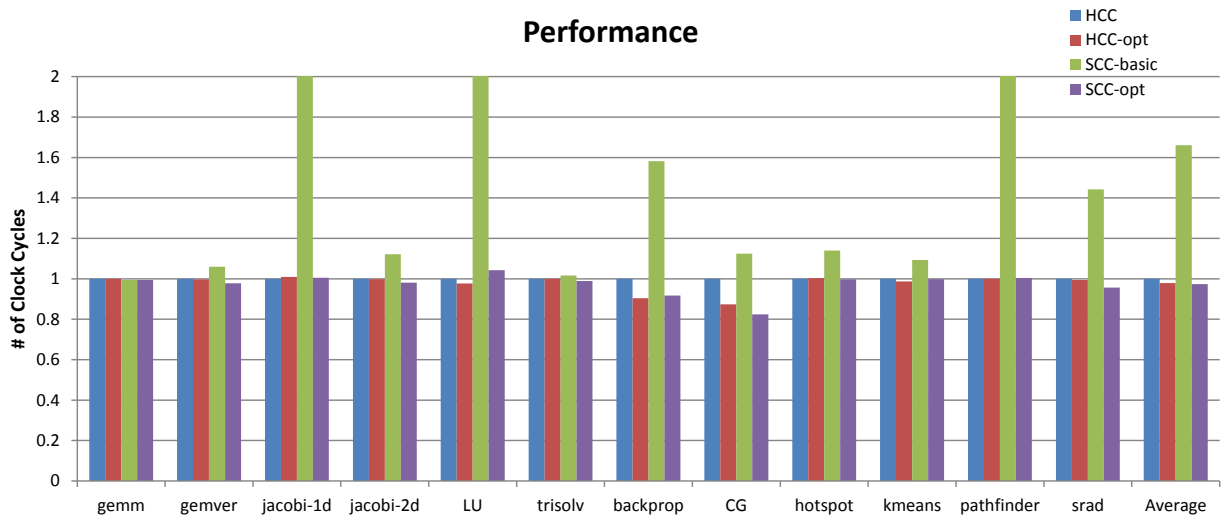

Figure A.12: Execution time (the lower, the better)

The running time (depicted in Figure A.12) shows a strong correlation between L1 cache read misses and performance. In HCC, the snooping overhead plays a significant role in determining execution time: In

our implementation, we assign 1 cycle to a read/write snooping request. In SCC, each coherence instruction incurs a two-cycle overhead. In addition to these overheads, there may be additional overheads depending upon the response to a snooping request in HCC (e.g., a read request may return an updated value from another processor) and the number of cache lines specified in the coherence instruction in SCC — each cache line incurs a 2-cycle delay. Because of removal of hardware cache coherence, we observe a 3% performance gain for SCC-opt over HCC on average.

Discussion: The performance results obtained for HCC and SCC schemes are sensitive to architectural choices made in the simulator implementation. And, we have opted for architectural choices that favor HCC even though on a real system they may be impractical or too costly. E.g., we have allotted a 1-cycle delay for a snooping request and on a real system it might take multiple cycles. The implemented HCC protocol in the simulator concurrently sends a snoop request to other cores, and also a memory request to the L2 cache. Alternately, the L1 cache can also be designed to send a memory request to the L2 cache after a snoop miss, but this will increase the delay when there is a snoop miss.

A.5.4 Energy Results

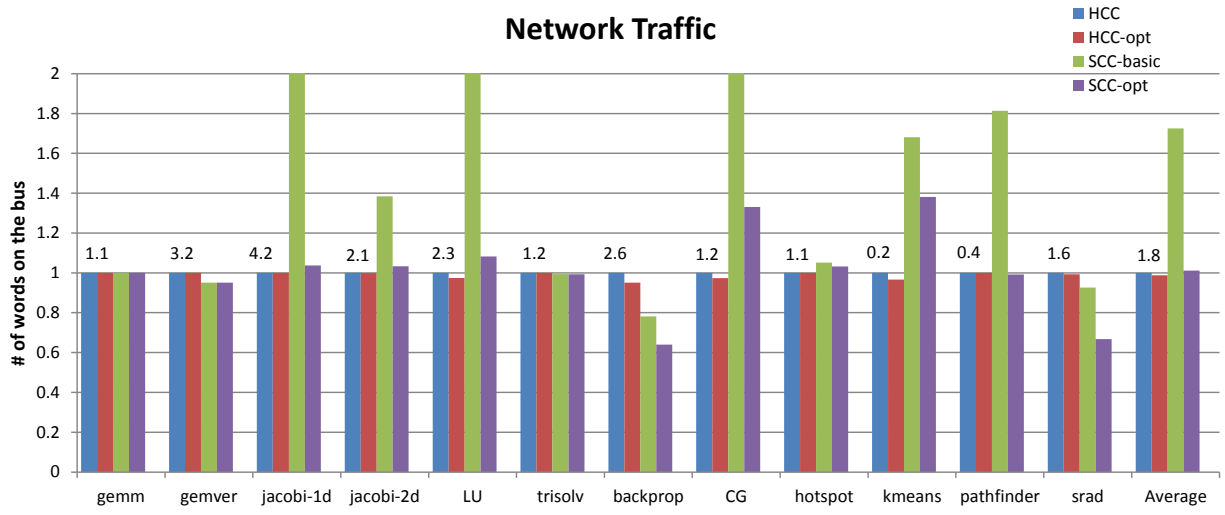

Figure A.13: Traffic on the system bus (the lower, the better). Average number of words per cycle for HCC is also shown above the bar.

Bus data transfers: Figure A.13 shows the traffic (number of words transferred) on the system bus for different schemes. All values are normalized with respect to HCC. The average number of words trans-

ferred per cycle (obtained by dividing total number of words with number of execution cycles) for HCC is also shown. For hardware coherence scheme, the traffic on the bus includes snoopy-bus coherence related exchange of messages, transfers between private L1 caches and shared L2 cache triggered by cache misses at L1 and replacement of cache lines at L1. For SCC, this includes data transfers between L1 caches and L2 cache prompted either by L1 misses and evictions, or invalidation and writeback coherence instructions. The HCC normalized data transfers on the bus for HCC-opt, SCC-basic, and SCC-opt are 0.99, 1.73, and 1.01 respectively, on average. In backprop and srad, SCC-opt does fewer write-backs to L2 cache compared to HCC; the L1 cache misses are lower for SCC-opt in the case of srad. Consequently, SCC-opt incurs fewer data transfers in backprop and srad. Conservative writebacks in kmeans increases the traffic on the bus for SCC-opt compared to HCC.

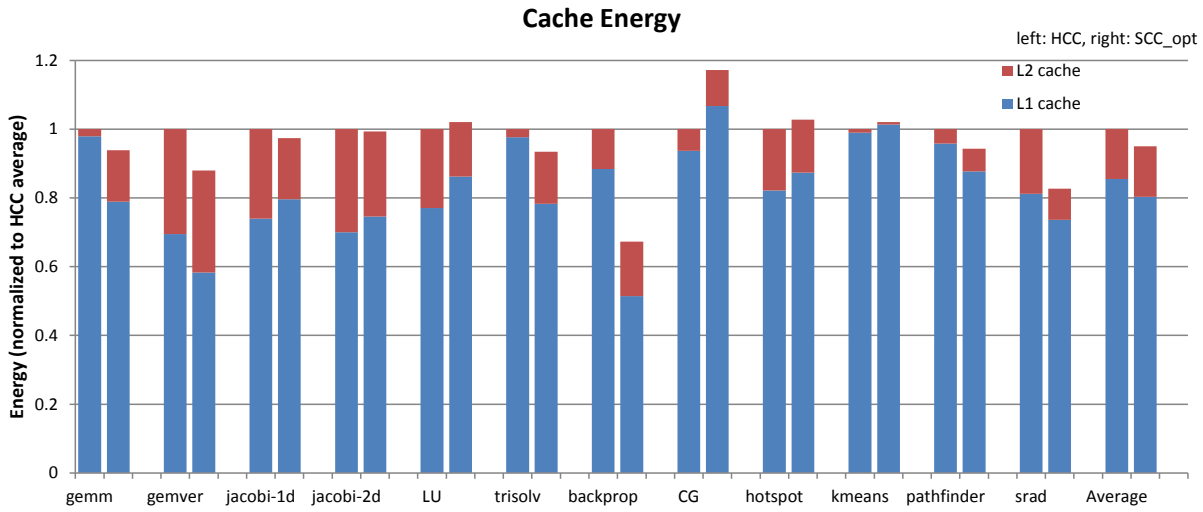

Figure A.14: L1 and L2 Cache Energy (the lower, the better). The first bar shows HCC energy and second bar SCC-opt energy

L1 and L2 cache energy: The cache SRAM is a major consumer of energy in a processor. We compare dynamic cache energy consumption for HCC and SCC-opt schemes based on the number of accesses to tag SRAMs and data SRAMs. Using the SESC simulator, event counts for all relevant activities in the L1 and L2 caches are collected to account for all tag and data accesses to SRAMS. CACTI [52] is used to obtain the energy per access for tag and data for each cache level. The L1 cache employs dual ported SRAM to service snoop requests quickly. For SCC also we used the same dual ported SRAM for a fair comparison (per-access cost is a function of, inter alia, number of ports). The L1 cache accesses tag and data together

for local processor requests while for snooping requests it accesses data SRAM only after tag hit. The L2 cache is configured to be in sequential access mode — it starts to access data SRAM after tag matching. We did not consider main memory energy because main memory accesses would be the same for both HCC and SCC schemes.

Figure A.14 plots relative energy consumption in caches for hardware and software cache coherence approaches: energy expenditure by HCC is considered 1 and energy dissipation by SCC-opt is scaled with respect to HCC. The break-down of energy expended in L1 and L2 caches is indicated. On average (arithmetic mean) SCC-opt energy consumption in caches is 5% less than that of HCC. Most of the savings in SCC-opt come from two sources: elimination of snooping requests in L1 cache, and reduction in the number of *writeback words* by partial line transfers (only dirty words are written back to shared L2 cache in a software managed cache as opposed to entire cache lines which are the granularity of communication for HCC). We also observe that energy spent in all L1 caches together is around 86%, while the rest — 14% is expended in L2 cache.

References

[1] DARPA UHPC program. http://www.darpa.mil/Our_Work/MTO/Programs/Ubiquitous_High_Performance_Computing_(UHPC).aspx.

[2] Open Community Runtime. https://01.org/open-community-runtime.

[3] QEMU. http://wiki.qemu.org/Main_Page.

[4] X-Stack Repository. https://xstack.exascale-tech.com/git/public/xstack.git.

[5] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *Computer*, 29(2):18–28, Feb. 1996.

[6] Arvind and D. E. Culler. Annual review of computer science vol. 1, 1986. chapter Dataflow Architectures, pages 225–253. Annual Reviews Inc., Palo Alto, CA, USA, 1986.

[7] T. J. Ashby, P. Diaz, and M. Cintra. Software-Based Cache Coherence with Hardware-Assisted Selective Self-Invalidations Using Bloom Filters. *Computers, IEEE Transactions on*, 60(4):472–483, Apr. 2011.

[8] A. Basumallik and R. Eigenmann. Towards Automatic Translation of OpenMP to MPI. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, pages 189–198, New York, NY, USA, 2005. ACM.

[9] A. Basumallik and R. Eigenmann. Optimizing irregular shared-memory applications for distributed-memory systems. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 119–128, New York, NY, USA, 2006. ACM.

[10] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT*, October 2008.

[11] B. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, B. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Polaris: The next generation in parallelizing compilers. In *PROCEEDINGS OF THE WORKSHOP ON LANGUAGES AND COMPILERS FOR PARALLEL COMPUTING*, pages 10–1. Springer-Verlag, Berlin/Heidelberg, 1994.

[12] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Taşirlar. Concurrent Collections. *Sci. Program.*, 18(3-4):203–217, Aug. 2010.

[13] J. B. Carter. Design of the Munin Distributed Shared Memory System. *Journal of Parallel and Distributed Computing*, 29(2):219–227, Sept. 1995.

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

[15] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE International Symposium on Workload Characterization*, 2009.

[16] H. Cheong and A. V. Vaidenbaum. A Cache Coherence Scheme with Fast Selective Invalidation. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, ISCA '88, pages 299–307, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[17] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou. DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 155–166, Washington, DC, USA, 2011. IEEE Computer Society.

[18] L. Choi and P.-C. Yew. A compiler-directed cache coherence scheme with improved intertask locality. In *Supercomputing*, 1994.

[19] R. Cledat, J. Fryman, I. Ganev, S. Kaplan, R. Khan, A. Mishra, B. Seshasayee, G. Venkatesh, D. Dunning, and S. Borkar. Functional simulator for exascale system research. In *Workshop on Modeling & Simulation of Exascale Systems & Applications*, 2013.

[20] E. Darnell, J. M. Mellor-Crummey, and K. Kennedy. Automatic software cache coherence through vectorization. In *International Conference on Supercomputing*, 1992.

[21] R. Das, M. Uysal, J. Saltz, and Y.-S. Hwang. Communication Optimizations for Irregular Scientific Computations on Distributed Memory Architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–478, Sept. 1994.

[22] J. B. Dennis. First version of a data flow procedure language. In *Programming Symposium, Proceedings Colloque Sur La Programmation*, pages 362–376, London, UK, UK, 1974. Springer-Verlag.

[23] M. Elver and V. Nagarajan. TSO-CC: Consistency directed cache coherence for TSO. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 165–176, Feb 2014.

[24] ET International, Inc. SWARM (SWift Adaptive Runtime Machine). `http://www.etinternational.com/index.php/products/swarmbeta/`.

[25] P. Feautrier. Dataflow Analysis of Array and Scalar References. *IJPP*, 20(1), 1991.

[26] A. Friedley, G. Bronevetsky, T. Hoefler, and A. Lumsdaine. Hybrid MPI: Efficient message passing for multi-core systems. In *High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for*, pages 1–11, Nov 2013.

[27] G. R. Gao, K. B. Theobald, A. Marquez, and T. Sterling. The HTMT Program Execution Model. Technical report, In Workshop on Multithreaded Execution, Architecture and Compilation (in conjunction with HPCA-4), Las Vegas, 1997.

[28] M. W. Hall, S. Amarasinghe, B. Murphy, S.-W. Liao, and M. Lam. Detecting Coarse-Grain Parallelism Using an Interprocedural Parallelizing Compiler. In *Supercomputing, 1995. Proceedings of the IEEE/ACM SC95 Conference*, pages 49–49, 1995.

[29] B. Hechtman, S. Che, D. Hower, Y. Tian, B. Beckmann, M. Hill, S. Reinhardt, and D. Wood. QuickRelease: A throughput-oriented approach to release consistency on GPUs. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 189–200, Feb 2014.

[30] A. Hoisie and V. Getov. Extreme-Scale Computing. In *Special Issue, IEEE Computer Magazine*, November 2009.

[31] Intel. X-Stack Traleika Glacier project. `https://xstack.exascale-tech.com/wiki/index.php/Main_Page`.

[32] S. Kaxiras and G. Keramidas. SARC Coherence: Scaling Directory Cache Coherence in Performance and Power. *IEEE Micro*, 30(5):54–65, Sept. 2010.

[33] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel. Rigel: An Architecture and Scalable Programming Interface for a 1000-core Accelerator. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 140–151, New York, NY, USA, 2009. ACM.

[34] J. H. Kelm, D. R. Johnson, W. Tuohy, S. S. Lumetta, and S. J. Patel. Cohesion: A Hybrid Memory Model for Accelerators. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 429–440, New York, NY, USA, 2010. ACM.

[35] P. Kogge et al. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. In *DARPA-IPTO Sponsored Study*, September 2008.

[36] L. Kontothanassis and M. Scott. Software cache coherence for large scale multiprocessors. In *High-Performance Computer Architecture, 1995. Proceedings., First IEEE Symposium on*, pages 286–295, 1995.

[37] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[38] C. Liao, D. J. Quinlan, T. Panas, and B. R. de Supinski. A ROSE-Based OpenMP 3.0 Research Compiler Supporting Multiple Runtime Libraries. In *Proceedings of the 6th International Conference on Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More*, IWOMP'10, pages 15–28, Berlin, Heidelberg, 2010. Springer-Verlag.

[39] M. F. P. O'Boyle, R. W. Ford, and E. A. Stohr. Towards General and Exact Distributed Invalidation. *Journal of Parallel and Distributed Computing*, 63(11):1123–1137, Nov. 2003.

[40] P. M. Ortego and P. Sack. SESC: SuperESCalar simulator. In *Euro micro conference on real time systems*, 2004.

[41] PoCC: the Polyhedral Compiler Collection. `http://sourceforge.net/projects/pocc/`.

[42] PolyBench: The Polyhedral Benchmark suite. `http://sourceforge.net/projects/polybench/`.

[43] D. Quinlan et al. ROSE Compiler Infrastructure. `http://www.rosecompiler.org`.

[44] A. Ros, M. Davari, and S. Kaxiras. Hierarchical Private/Shared Classification: the Key to Simple and Efficient Coherence for Clustered Cache Hierarchies. In *21st Symposium on High Performance Computer Architecture (HPCA)*, Bay area, CA (USA), Feb. 2015. IEEE Computer Society.

[45] A. Ros and S. Kaxiras. Complexity-effective Multicore Coherence. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 241–252, New York, NY, USA, 2012. ACM.

[46] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A Low Overhead, Software-only Approach for Supporting Fine-grain Shared Memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, pages 174–185, New York, NY, USA, 1996. ACM.

[47] I. Singh, A. Shriraman, W. Fung, M. O'Connor, and T. Aamodt. Cache coherence for GPU architectures. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 578–590, Feb 2013.

[48] H. Sung and S. V. Adve. DeNovoSync: Efficient Support for Arbitrary Synchronization Without Writer-Initiated Invalidations. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 545–559, New York, NY, USA, 2015. ACM.

[49] H. Sung, R. Komuravelli, and S. V. Adve. DeNovoND: Efficient Hardware Support for Disciplined Non-determinism. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 13–26, New York, NY, USA, 2013. ACM.

[50] K. B. Theobald. *Earth: An Efficient Architecture for Running Threads*. PhD thesis, Montreal, Que., Canada, Canada, 1999. AAINQ50269.

[51] S. Verdoolaege. isl: An integer set library for the polyhedral model. *Mathematical Software–ICMS 2010*, pages 299–302, 2010.

[52] S. J. E. Wilton and N. P. Jouppi. CACTI: An Enhanced Cache Access and Cycle Time Model. *IEEE Journal of Solid-State Circuits*, 31, 1996.

[53] J. Wu, R. Das, J. Saltz, H. Berryman, and S. Hiranandan. Distributed memory compiler design for sparse problems. *Computers, IEEE Transactions on*, 44(6):737–753, Jun 1995.

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