Architecting and Programming a Hardware-Incoherent Multiprocessor Cache Hierarchy

Wooil Kim, Sanket Tavarageri,[†] P. Sadayappan,[†] and Josep Torrellas

University of Illinois, Urbana-Champaign
http://iacoma.cs.uiuc.edu

[†]Ohio State University
tavarageri.1,sadayappan.1@osu.edu

***Abstract*—**
New architectures for extreme-scale computing need to be designed for higher energy efficiency than current systems. One recently-proposed extreme-scale manycore radically simplifies the architecture, and proposes a cluster-based on-chip memory hierarchy without hardware cache coherence. To program for such an environment, this paper proposes two approaches. They use shared-memory programming either inside clusters only, or both inside and across clusters. Both approaches rely on ISA support for writeback and self-invalidation operations. Our simulation results show that hardware-incoherent cache hierarchies with our support deliver reasonable performance for applications that were not written for such hierarchies. Specifically, for execution within a cluster, the average execution time of the applications is 2% higher than with hardware cache coherence; for execution across multiple clusters, it is 5% higher than with hardware cache coherence. This is accomplished with minimal hardware support.

***Keywords*-Cache coherence; Hardware-incoherent caches; Software-managed caches.**

I. Introduction

Continuous progress in transistor integration is delivering manycores with ever-increasing transistor counts. In the next few years, to be able to utilize all the cores in these chips effectively — for example, to build Exascale machines [1] — these chips 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 [2], [3].

One such extreme-scale architecture is Runnemede [4]. 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 so-called hardware-incoherent caches [4].

A cache hierarchy without hardware coherence has obvious advantages in ease of chip implementation. In addition, it enables a more energy-efficient execution of some of the applications expected to run on these machines. However, it presents a non-trivial programming challenge. Indeed, some software has to orchestrate the movement of data between the different caches and cache levels. Moreover, although one can argue that large exascale machines such as a Runnemede 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 cache hierarchy.

There are several other proposals of multiprocessor cache hierarchies without hardware coherence or with simplified hardware coherence. Two examples are Rigel [5] and Cohesion [6], which are manycores designed to accelerate regular visual programs. There is also substantial work on software cache coherence schemes, mostly relying on detailed compiler analysis of the code (e.g., [7], [8], [9]), and also using bloom filters to summarize communication [10]. Moreover, there are efforts that try to simplify the design of hardware cache coherence protocols (e.g., [11], [12]).

Building on this past work, our goal is to design a user-friendly programming environment to exploit a cluster-based hardware-incoherent cache hierarchy like Runnemede's. We are not seeking to simplify or minimize hardware cache coherence protocols. Instead, we attempt to exploit this easy-to-build and unconventional cache hierarchy using, as much as possible, existing cache structures and programming styles.

The contributions of this paper are as follows:
• We propose simple hardware extensions to manage a hardware-incoherent cache hierarchy. They are several flavors of writeback and self-invalidation instructions, two small buffers next to the L1 cache, and a hardware table in the cache controller.
• We introduce two relatively user-friendly programming approaches for the machine that use our hardware extensions. These programming approaches involve shared-memory programming either inside clusters only, or both inside and across clusters. They rely on annotating synchronization operations, and on identifying producer-consumer pairs.
• Simulation results showing that our programming approaches allow hardware-incoherent cache hierarchies to deliver reasonable performance for applications that were not written for incoherent hierarchies. Specifically, 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, it is only 5% higher than with hardware cache coherence. This is accomplished with our minimal hardware support.

This paper is organized as follows. Section II motivates the work further; Section III describes the basic ISA used; Sections IV and V present our two programming approaches and additional hardware extensions; Sections VI and VII evaluate our design; and Section VIII discusses related work.

II. Motivation

Runnemede [4] is an extreme-scale manycore recently proposed 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 shows the manycore's architecture from [4].

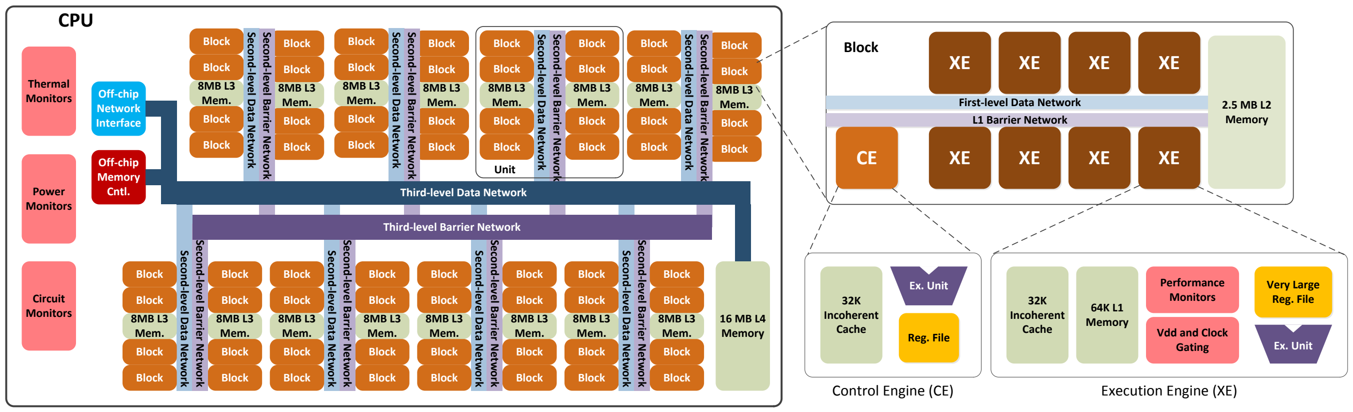

Figure 1: The Runnemede chip architecture from [4].

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

In Runnemede, L2 and L3 are memories rather than caches, each with their own range of assigned addresses. In this paper, however, we will use them as successive levels of on-chip incoherent caches, to better resemble a conventional machine.

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 an opportunity 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 paper, we focus on proposing simple hardware extensions and programming models for a cluster-based hardware-incoherent cache hierarchy like the one described.

III. Basic Architectural Support

A. Using Hardware-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 there is 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 shows the sequence.

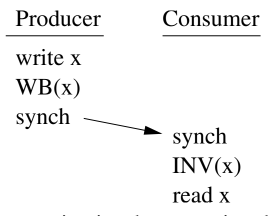

Figure 2: Communication between incoherent caches.

We call the set of dynamic instructions between two consecutive synchronizations 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 has 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.

B. Instructions for Coherence Management

WB and INV are memory instructions that give commands to the cache controller. For this initial discussion, assume that each core has a private L1 cache and that there is a shared L2 cache. WB and INV can have different flavors. First, they can be invoked with different data granularities: byte, half word, word, double word, or quad word. In this case, they take as an argument the address of the operand. Second, WB and INV can also operate on a range of addresses. In this case, they take as arguments the start address and the length of the range. In addition, 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. For example, if our finest grain for sharing is a word (as we will assume in our discussion), then we use per-word Dirty (D) bits. When a WB is executed, the cache controller reads the target line(s) and only writes back the dirty words. WB has no effect if the target addresses contain no dirty valid data. After a line is written back, it is left in state clean valid 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 data, the cache controller first writes it back to the shared cache before invalidating the line.

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 data moves.

C. Instruction Reordering

The INV and WB instructions introduce some reordering constraints with respect to loads and stores to the same address. The required constraints are that neither the compiler nor the hardware can reorder *INV(x)→ld x* (Figure 3a), or *st x→WB(x)* (Figure 3b). Effectively, a thread uses *INV(x)* to refresh its view of *x* and, therefore, reordering it with a subsequent load would imply that the load could fail to see the new value that the program intended. Similarly, a thread uses *WB(x)* to globally post the value of *x* and, therefore, reordering it with a prior store would imply that the value posted would be different than the one the program intended.

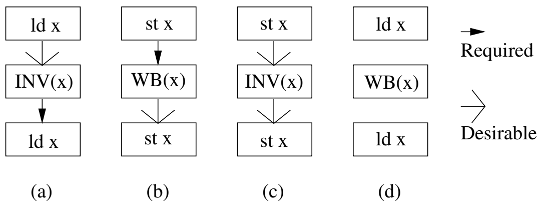

Figure 3: Ordering constraints.

Figure 3 also shows some desirable orders that both compiler and hardware should retain. Consider *ld x→INV(x)* (Figure 3a). Reordering the accesses is likely to cause additional traffic because the load will now miss in the cache. Moreover, a core may execute a load followed by INV in a spinning loop; reordering the accesses is likely to delay when the core will see a new global value.

Consider now *WB(x)→st x* (Figure 3b). A producer core may execute a WB(x) followed by a store to x in a loop, to make sure that new values are posted as soon as they are generated; reordering the accesses is likely to change when values are globally posted.

The case *st x→INV(x)→st x* (Figure 3c) is likely unusual. However, it is desirable to enforce both orders for the same reason as above: if these accesses are in a loop, reordering them changes when values are globally posted.

Finally, consider loads that precede or follow a WB (Figure 3d). Such accesses can always be reordered because a WB does not change the value of the line in the local cache. Hence, a reodered load sees the same value. In fact, the ability to pick a load that follows a WB to the same address and execute it before the WB can help performance by prefetching data.

Current processor hardware naturally enforces most of these reordering constraints. This is because we envision WB and INV to proceed in the pipeline like stores, and be deposited in the write buffer at retirement time like stores. In the write buffer, stores to the same location must be drained in order. Hence, given a *WB(x)* or *INV(x)* in the write buffer, prior or subsequent stores to *x* would not be able to reorder relative to them. In addition, loads to *x* before *WB(x)* or *INV(x)* will not be reordered because the loads will be finished by the time *WB(x)* or *INV(x)* retire into the write buffer.

We envision the new pipeline hardware to be designed such that loads to *x* are not allowed to bypass an earlier *INV(x)*; however, for performance, they are allowed to bypass an earlier *WB(x)* and proceed directly to the write buffer and cache (Figure 3d).

D. 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 Tera [13], IBM RP3 [14], or Cedar [15] have provided special hardware synchronization in the memory subsystem. Such hardware often queues-up 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 [4] 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 controller of the shared cache 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. Since synchronization support is not the focus of this paper, we do not consider further details.

IV. 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 message sender and a message receiver communicate by writing to and reading from an on-chip uncacheable shared buffer. Of course, sender and receiver need to synchronize to ensure that writes and reads happen in the right sequence. Moreover, the library needs to handle buffer overflows. In communication with multiple recipients such as a broadcast, there is no need to make multiple copies; the sender only needs to perform a single write. The multiple receivers will all read from the same location. Finally, we can implement the nonblocking MPI_Isend and MPI_Irecv calls by using another thread in the core to perform the writes and reads. A possible implementation is presented by Friedley et al. [16].

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

A. 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

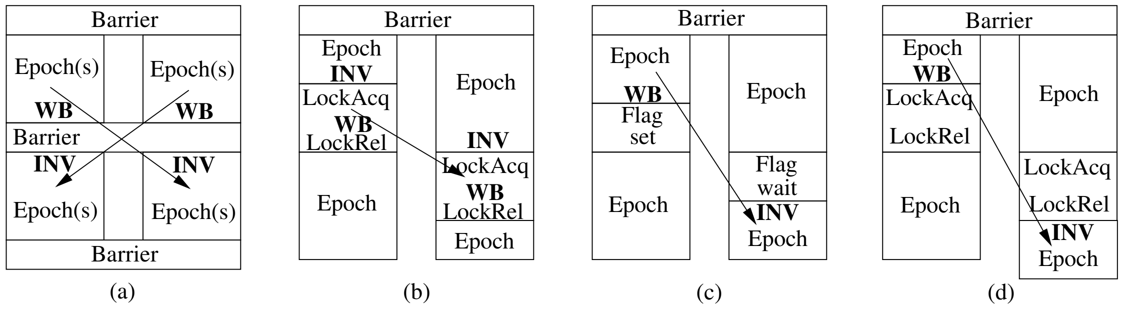

Figure 4: Annotations for communication patterns enabled by: barriers (a), critical sections (b), flags (c), and dynamic happens-before epoch orderings (d).

many programs, however, the analysis would be too conservative, leading to bad performance. 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 the performance.

The approach is based on relying on the synchronization operations in the program as explicit markers that separate data dependences between threads. 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. Of course, we need to consider data races separately, as they are data dependences that are not ordered by synchronization operations.

Recall that a block has a per-core private L1 cache and a shared L2 cache (Figure 5a). In addition, there is a one-to-one thread-to-core mapping and no thread migration.

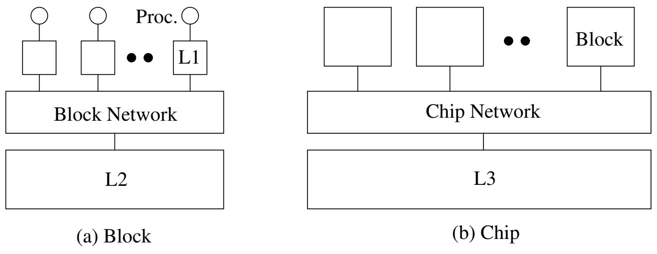

Figure 5: Single block (a) and multi-block (b) cache hierarchy.

1) Annotating Different Communication Patterns: Each type of synchronization operation tells us 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 4a). 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 codes, a thread owns part of the shared space, and reuses it across the barriers as if it was private. In this case, we do not write back or 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 sharing pattern information, we can use WB ALL and INV ALL, which write back and invalidate the whole L1 cache.

Threads often communicate by accessing variables inside a critical section protected by lock acquire and lock release (Figure 4b). In this case, after we enter a critical section, we need to eliminate any stale data from the cache. Moreover, before we leave, we need to post all the updates created in the critical section. Consequently, immediately after the acquire, we would place an INV for all the shared variables that will receive exposed reads inside the critical section. In addition, immediately before the release, we place a WB for all the shared variables written inside the critical section. To reduce the duration of the critical section, however, we place the INV *immediately before* the acquire rather than *after* it. This is correct because we assume that the cache cannot change state between the INV and the acquire — i.e., there is no context switch to another thread that could pollute the cache.

Threads also coordinate with flag set-wait synchronization (Figure 4c). In this case, immediately before a thread sets the flag, the 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 a thread successfully completes the wait, the 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. Specifically, after a thread completes a critical section, it may want to consume data that was generated by earlier holders of the critical section lock before they executed the critical section (Figure 4d). 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.

2) 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.

Some programs may contain data races. In this case, relying on explicit synchronization to orchestrate data transfers is insufficient. Indeed, the communication that data races induce in cache-coherent hierarchies does not occur in a hardware-incoherent cache hierarchy. For example, assume that two processors try to communicate with a store and a spinloop on a variable *flag* that is declared volatile (Figure 6a). In an incoherent cache hierarchy, the consumer may never see the update.

```
// producer     // consumer

 data = 1;      for (; ; ) {
 fence;           if (flag) {
 flag = 1;           fence;
                     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 6: Enforcing data-race communication.

If we want to enforce the data-race communication, we need to augment the write and read of *flag* with WB and INV, respectively. In addition, to ensure that the effect of the fence in a processor is observed by the other processor, we need WB and INV for the data passed (*data*) as well (Figure 6b). If the program can be re-written, a better solution is to eliminate the race by re-writing the code to use synchronization or, in C++11, declare *flag* as atomic.

3) Application Classification: Table I lists the applications that we use for intra-block programming in Section VII, 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 can be those just described: barrier, critical section, flag, outside critical section, and data race. Cholesky had busy-waiting on variables; to reduce unnecessary traffic, we changed it to flag synchonization.

Appl.	Main	Other
FFT	Barrier	
LU	Barrier	
Cholesky	Outside critical	Barrier, critical, flag
Barnes	Barrier, outside critical	Critical
Raytrace	Critical	Barrier, data race
Volrend	Barrier, outside critical	
Ocean	Barrier, critical	
Water	Barrier, critical	

Table I: Communication patterns observed in our applications for intra-block programming.

From the table, we see that different applications have different dominant communication patterns. In addition, some applications have multiple patterns. Overall, the data shows that our programming model needs to support at least all the communication patterns that we described earlier.

B. 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. 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 short epochs such as critical sections.

1) 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 word is updated (assuming that the cache has per-word 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 and only those. 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.

2) 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 minimize the number invalidations.

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 starts the epoch empty. The IEB is accessed at every L1 read. If the line's address is already in the IEB, or the read hits in the cache and the target word is dirty (again, assuming per-word dirty bits), no special action is taken. The latter case occurs when the word was written in the past by the current core and, therefore, it is not stale. In all other cases, the hardware does the following: (1) the line's address is added to the IEB; (2) if the read hits in the cache, the cache line is first invalidated as it is considered the first read in the epoch; and (3) the read gets a fresh copy of the line from the shared cache into the L1 and a fresh copy of the word to the processor. Unlike the MEB, the IEB does not store unneeded entries.

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. Invalidations are expensive because they are followed by a cache miss in the critical path.

V. 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 cores — irrespective of whether they are in the same block or in different ones (Figure 5b). Our 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 implicitly operate as described until now: WB writes back the line to L2 and INV invalidates the line from L1. However, if the two threads end-up being mapped to different blocks, then WB implicitly writes back the line all the way to L3, and INV implicitly 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 completion, we also provide instructions that write back or invalidate lines to/from a given cache level. For example, *WB_L3(Addr)* writes back *Addr* to the L3 cache (and to the L2 in the process), and *INV_L2(Addr)* invalidates *Addr* from the L2 cache (and from the L1 in the process).

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

A. Inter-block Programming

In the programming approach of Section IV, 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. 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 cores at runtime (i.e., which cores 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*.

To understand how the instructions are used, assume that thread i produces x and thread j consumes it. Figure 7 shows the instructions inserted. Specifically, 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 instructions's arguments imply how the data should be transferred.

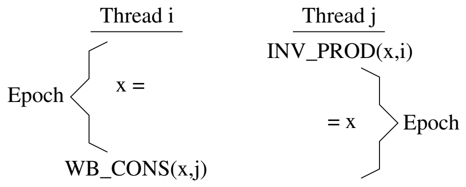

Figure 7: Example of level-adaptive WB and INV.

Generating this information requires deeper code analysis than in the first programming model (Section IV). Hence, while the first programming model can handle applications with pointers and irregular structures, this second model is targeted to more compiler-analyzable codes.

To be able to insert level-adaptive WB and INV instructions, the compiler starts by performing interprocedural 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 our compiler algorithm for level-adaptive WB and INV. First, we present the general case; then we consider irregular applications.

1) 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 dependences between serial and parallel sections, and between parallel sections. Typically, the relationship between serial and parallel sections is relatively easy to analyze, while the relationship between parallel sections is harder to. It requires understanding the program's inter-procedural control flow and performing data flow analysis.

Inter-procedural control flow analysis finds parallel *for* loops that potentially communicate with each other. Starting from each *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 data flow 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 array 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 epoch and INV_PROD (address, producerID) in the consumer epoch.

We perform this analysis using some extensions to the ROSE compiler infrastructure [17] that we developed. The consumer and producer IDs in the WB_CONS and INV_PROD instrumentations are typically expressed as equations. Our approach is similar to the translation from OpenMP to MPI [18] and to the implementation of software DSM [19]. 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 these two other techniques in four aspects. First, our approach does not maintain explicit replicated copies of data, while translated MPI code and software DSM do. 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. Finally, our approach is reasonably efficient even when exact data flow 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.

2) 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 [20] 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 8 shows an excerpt from a conjugate gradient program. The original code contained a loop that reads array *p[]* with indirect accesses (Line 23), and another loop that writes *p[]* (Line 29). We assume static scheduling of OpenMP loops with chunk distribution. Thus, each thread gets a set of contiguous iterations.

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

Figure 8: An iterative loop with irregular data accesses.

The inspector loop is placed at the beginning (Line 9). It determines the ID of the writer thread that will produce the value obtained by each read. The result is stored in array *conflict*. Then, at every iteration of the loop that reads, the code checks the ID of the writer (Line 21). If it is not the same as the reader's ID, it inserts an INV_PROD instruction with the ID of the thread that will produce the data (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 of the whole range written. We could perform an analysis like the one used for the read, and save some writes to L3 by using WB_CONS. To reduce the complexity of the analysis, however, we write everything to L3.

B. Hardware Support

We implement WB_CONS and INV_PROD as follows. In each block, the L2 cache controller has a *ThreadMap* hardware 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.

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 words only to L2. Otherwise, the dirty words 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 may require checking both the L1 and L2 tags.

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, WB_CONS ALL writes back not just the local L1 but also the whole local block's L2 to the L3. Similarly, in this case, INV_PROD ALL 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.

VI. Experimental Setup

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

Consider the intra-block runs first. We evaluate the architecture configurations in the upper part of Table II. For programming simplicity, the baseline uses WB ALL and INV ALL for all the synchronization primitives of Section IV-A and shown in Figure 4. Then, *B+M*, *B+I*, and *B+M+I* augment the baseline with the MEB, IEB, and both MEB and IEB, respectively. These hardware buffers are only used in critical sections. Finally, we compare to the same machine with hardware cache coherence (*HCC*).

We run the SPLASH-2 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 [21] to model the architecture in the upper part of Table III.

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 to L3; INV ALL from L2
Addr	WB of addresses to L3; INV of addresses from L2
Addr+L	WB_CONS and INV_PROD
HCC	Hardware cache coherence

Table II: Configurations evaluated.

The architecture has 16 cores in a block. The coherent architecture (*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
Per-core MEB	16 entries. Size: 9b (ID) + 1b (Valid)
Per-core IEB	4 entries. Size: 40b (Line addr) + 1b (Valid)
Shared L2	One bank per core. Each bank: 128KB WB, 8-way, 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 in 4 banks. Each bank: 4MB WB, 8-way, 20-cycle RT (local bank), 64B lines

Table III: Architecture modeled. RT means round trip.

We now consider the inter-block runs. We evaluate the configurations in the lower part of Table II. 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 also communicates via L3 but specifies the addresses to be written back or invalidated. Finally, *Addr+L* uses our *level-adaptive* WB and INV instructions, which transparently pick the correct cache level to write back and invalidate data to/from. In addition, it also specifies the addresses to operate on. The coherent machine (*HCC*) uses a hierarchical full-mapped directory-based MESI protocol.

We cannot run the applications used in intra-block experiments because the compiler cannot analyze them. Instead, we run four OpenMP applications, namely EP, IS and CG from the NAS Parallel Benchmark suite, and a 2D Jacobi application that we developed. CG is irregular. We use an analysis tool that we developed based on the ROSE compiler [17] 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. We do not apply any loop optimizations such as loop interchange or loop collapse.

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

VII. Evaluation

We first compare the control and storage overhead of hardware-incoherent and coherent cache hierarchies, then evaluate the intra-block hardware and programming model, and finally evaluate the inter-block hardware and programming model.

A. Control and Storage Overhead

We consider our hierarchical architecture with 4 blocks of 8 cores each, and compare the overhead needed by the incoherent and the coherent cache hierarchies discussed. Both hierarchies require special control and storage structures. In terms of control, coherent hierarchies require the cache coherence controllers, which are possibly associated with each cache controller. Incoherent hierarchies require much simpler control, namely the ability to write back and self-invalidate cache lines, and the per-L2 ThreadMap table.

In terms of storage structures, coherent hierarchies require the directory, and the coherence state bits in the L1 and L2 cache lines. Incoherent hierarchies require the per-core MEB and IEB buffers and, in each L1 and L2 cache line, a valid bit and per-word dirty bits. The L3 cache is similar in both systems.

Consider the coherent hierarchy first. To estimate the directory size, we assume a hierarchical, full-mapped directory protocol. Each L3 line needs 4 presence bits (since we have 4 blocks) and a dirty bit, while each L2 line needs 8 presence bits and a dirty bit. A MESI protocol has 4 stable coherence states and several transient ones. Hence, we assume that we need 4 bits to encode the coherence state in each L1 and L2 line.

In the incoherent hierarchy, the per-core MEB and IEB buffers have a very small size (Table III). Each L1 and L2 cache line needs a valid bit and per-word dirty bits. Given our cache line size, this is 16 dirty bits per line.

Overall, adding-up the sizes of all of these storage structures for the 32-core machine considered, we find that the hardware-incoherent hierarchy uses about 102KB less storage than the coherent one. This is a very small savings in storage.

B. Performance of Intra-block Execution

Figure 9 shows the execution time of the applications for the different architectures considered. For each application, 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 thread load imbalance. Lock stall time is the time spent waiting for lock acquires. Part of this time is caused by WB stalls, which delay the end of critical sections.

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, in these codes, all the architectures considered have performance similar to *HCC*.

The rest of the applications have finer-grain synchronization. In these codes, in *Base*, the overhead of WB and INV stalls, or their impact on *lock stall* is large. In Raytrace, there are frequent lock accesses in a set of job queues. Its fine-grain structure is the reason for the large overhead. The average bar shows that the majority of the overhead in *Base* comes from *lock stall*, which is largely the result of *WB stall*. On average for all the applications, *Base*'s execution time is 20% higher than *HCC*.

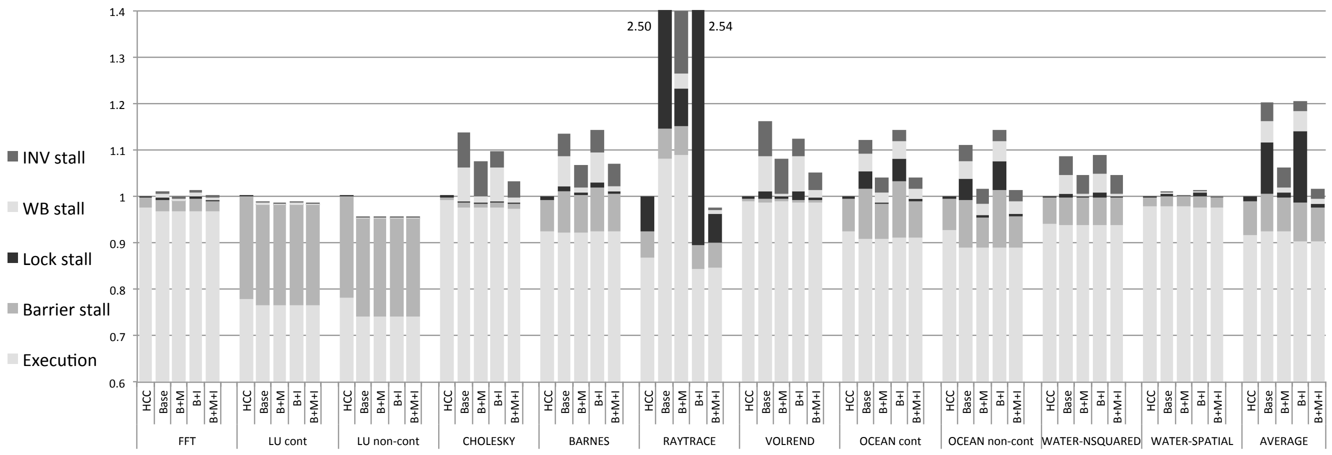

Figure 9: Normalized execution time of the applications.

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 most of the *WB stall* and *lock stall*. We can see, therefore, that the MEB is effective. The resulting bars (*B+M*) are close to *HCC*. One exception is Raytrace, where the bar is still high.

The third bar (*B+I*) shows that the IEB alone is not very effective at speeding-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 different traffic gives us some idea of their different energy consumption. Figure 10 shows, for each application, their relative network traffic in number of 128-bit 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 L1 and L2 caches: linefill due to read/write misses, writeback traffic, and invalidations.

From the figure, we see that these applications have on average 4% 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,

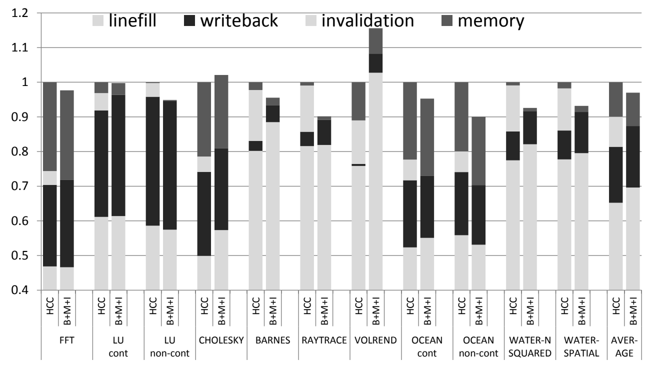

Figure 10: Normalized traffic of the applications.

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 actually written for an incoherent cache hierarchy, the total traffic and energy 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.

C. Performance of Inter-block Execution

To evaluate the effect of level-adaptive WB and INV, we start by counting the number of global WBs (those going to L3) and global INVs (those going to L2). We evaluate the *Addr* and *Addr+L* configurations from Table II. Recall that *Addr* always performs global WBs and INVs, while *Addr+L* only performs them if producer and consumer cannot be proven to be in the same block. Figure 11 compares these counts for our applications. The bars are normalized to the values for *Addr*.

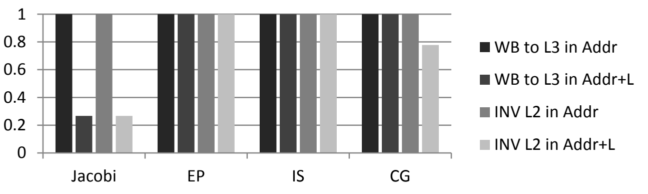

Figure 11: Normalized number of global WB and INV.

From the figure, we see that Jacobi and CG take advantage of level-adaptive instructions (*Addr+L* bars). In Jacobi, the number of global WBs and INVs remaining is 25% of the number in *Addr*. In CG, the number of global INVs remaining is 78%; to eliminate global WBs requires a more complicated compiler analysis. Overall, with these level-adaptive instructions (WB_CONS and INV_PROD), we reduce network traffic and miss 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 determine producer-consumer pairs. 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 12 compares the performance of all the configurations in the lower part of Table II (*HCC*, *Base*, *Addr*, and *Addr+L*). The bars are normalized to *HCC*. Recall that *Base* always performs WB ALL to L3 and INV ALL to L2. We see that *Base* performs the worst. In Jacobi, knowing which addresses to WB and INV (*Addr*) pays off. In CG, making some of the INVs local with *Addr+L* has a further performance impact. In EP and IS, the reduction operations prevent *Addr* and *Addr+L* from having any benefits over *Base*.

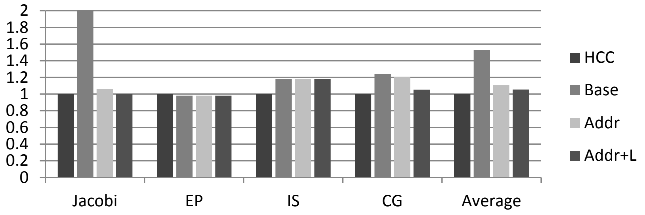

Figure 12: Normalized execution time of the applications.

On average, *Addr+L* reduces the execution time by 5% over *Addr* and by 31% over *Base*. We conclude that level-adaptive WB and INV can speed-up codes by transforming some global operations into intra-block ones. However, the impact is a strong function of the application — in particular, the contribution of the global memory operations to the execution time, and the analyzability of the memory operations.

Figure 12 also shows that *HCC* is faster than *Addr+L*. The reason is that, in *Addr+L*, the WB and INV instructions are sometimes conservative. In addition, the latency of WB and INV instructions is often hard to hide. On average, however, *Addr+L* only takes 5% more time to execute. This is tolerable, especially for applications that were not specifically written for an incoherent cache hierarchy.

VIII. Related Work

The most closely related works are Rigel [5] and its successor Cohesion [6]. While these works also use hardware-incoherent caches, and writeback and invalidation operations, we have three contributions over these past works. First, we examine the ordering constraints between the INV and WB instructions, and other accesses. Second, we provide a simple yet effective methodology to insert writebacks and invalidations both within a block and across blocks. Third, we propose the use of level-adaptive writebacks and invalidations to attain performance in a multi-block machine. Rigel's writebacks and invalidations always direct communication through the last level cache.

Runnemede [4] does not specify how to communicate between blocks except through DMA operations initiated by a DMA engine. Rigel and Cohesion rely on manual coding for inter-block communication. Communication occurs through the last-level global cache regardless of the current thread mapping. This inefficiency is justified by their focus on accelerator workloads, which do not have much inter-block communication. However, more communication-oriented algorithms, or programs that access memory in an irregular manner do not perform efficiently on those architectures.

Our work is related to software cache coherence schemes, which typically rely on compiler analysis of the code (e.g., [7], [8], [9]), or on using bloom filters to summarize communication [10]. One of the first works is Cheong et al. [7], who use data flow analysis to classify every reference to shared memory either as a memory-read or a cache-read. Invalidation is done selectively. Data flow analysis is carried out at the granularity of arrays, which may cause invalidations for an entire array. Choi et al. [8] propose to improve inter-task locality in software managed caches by using additional hardware support. Specifically, epoch numbers are maintained at runtime, and cache words are associated with them. An epoch flow graph technique establishes conditions under which it can be guaranteed that the cached copy of a variable is not stale. Darnell et al. [9] 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. Compared to these works, our intra-block approach takes a simpler approach based on relying on synchronizations. Our inter-block approach uses compiler techniques, but focuses on producer-consumer pairs and inspector-executor patterns.

Ashby et al. [10] support software-based cache coherence with selective self-invalidations using bloom filters. 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 can incur significant overhead in lock-intensive programs. In our proposal, we provide the MEB/IEB structures to isolate critical sections from other epochs.

Our reliance on identifying synchronization in the program is related to earlier work on lazy release consistency [22], performed in the context of Distributed Shared Memory (DSM) (e.g., as in TreadMarks [19]). DSM has similarities in that it provides a coherent memory image without direct hardware support for cache coherence.

Our work is also related to efforts that try to simplify the design of hardware cache coherence protocols (e.g., [11], [12], [23], [24]). The VIPS-M protocol [12] 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 write-through. Our work differs in that we use writeback L1 caches, and try to minimize unnecessary self-invalidation using MEB/IEB. DeNovo [11], [23] tries to simplify the hardware cache coherence protocol by using programming-model or language-level restrictions for shared memory accesses (e.g., determinism or data-race freedom). The benefit of DeNovo's approach is a simpler coherence protocol. In our work, we do not introduce any requirements to the programming model. Our approach takes and instruments legacy parallel codes. These works also differ from our work in that they are limited to a two-level cache hierarchy. Concurrent to our work, Ros et al. [24]

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, in order to infer the target of inter-thread communication.

IX. Conclusion

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 program for such hardware-incoherent cache hierarchy, this paper proposed simple hardware extensions and two user-friendly programming approaches. The hardware extensions are several flavors of writeback and self-invalidation instructions, two small buffers next to the L1 cache, and a table in the cache controller. The programming approaches involve shared-memory programming either inside clusters only, or both inside and across clusters.

Our simulation results showed that hardware-incoherent cache hierarchies with our support deliver reasonable performance for applications that were not written for incoherent hierarchies. Specifically, for execution within a cluster, the average execution time of the applications was only 2% higher than with hardware cache coherence; for execution across multiple clusters, it was only 5% higher than with hardware cache coherence. This was accomplished with our minimal hardware support.

Acknowledgment

This work was supported by NSF under CCF-1012759 and CCF-1536795, and by DOE ASCR under award DE-SC0008717.

References

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

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

[3] J. Torrellas, "Extreme-Scale Computer Architecture: Energy Efficiency from the Ground Up," in *International Conference on Design, Automation and Test in Europe (DATE)*, March 2014.

[4] 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 *International Symposium on High Performance Computer Architecture*, Feb. 2013.

[5] 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 *Int. Symp. on Comp. Arch.*, June 2009.

[6] J. H. Kelm, D. R. Johnson, W. Tuohy, S. S. Lumetta, and S. J. Patel, "Cohesion: A hybrid memory model for accelerators," in *International Symposium on Computer Architecture*, June 2010.

[7] H. Cheong and A. V. Veidenbaum, "A cache coherence scheme with fast selective invalidation," in *International Symposium on Computer Architecture*, June 1988.

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

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

[10] T. J. Ashby, P. Diaz, and M. Cintra, "Software-based cache coherence with hardware-assisted selective self-invalidations using bloom filters," *IEEE Trans. Comput.*, vol. 60, no. 4, pp. 472–483, Apr. 2011.

[11] 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 *International Conference on Parallel Architectures and Compilation Techniques*, Oct. 2011.

[12] A. Ros and S. Kaxiras, "Complexity-effective multicore coherence," in *International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2012.

[13] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, "The Tera Computer System," in *International Conference on Supercomputing*, 1990, pp. 1–6.

[14] G. Pfister *et al.*, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," in *International Conference on Parallel Processing*, 1985, pp. 764–771.

[15] D. Kuck *et al.*, "The Cedar System and an Initial Performance Study," in *International Symposium on Computer Architecture*, 1993, pp. 213–223.

[16] A. Friedley, G. Bronevetsky, T. Hoefler, and A. Lumsdaine, "Hybrid MPI: Efficient Message Passing for Multi-core Systems," in *Supercomputing*, Nov. 2013.

[17] ROSE Compiler Infrastructure, www.rosecompiler.org.

[18] A. Basumallik and R. Eigenmann, "Towards Automatic Translation of OpenMP to MPI," in *International Conference on Supercomputing*, June 2005.

[19] C. Amza *et al.*, "TreadMarks: Shared Memory Computing on Networks of Workstations," *Computer*, vol. 29, no. 2, pp. 18–28, Feb. 1996.

[20] R. Das, M. Uysal, J. Saltz, and Y.-S. Hwang, "Communication optimizations for irregular scientific computations on distributed memory architectures," *J. Parallel Distrib. Comput.*, 1994.

[21] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos, "SESC simulator," January 2005, http://sesc.sourceforge.net.

[22] P. Keleher, A. Cox, and W. Zwaenepoel, "Lazy Release Consistency for Software Distributed Shared Memory," in *International Symposium on Computer Architecture*, May 1992.

[23] H. Sung, R. Komuravelli, and S. V. Adve, "DeNovoND: Efficient Hardware Support for Disciplined Non-determinism," in *Conf. on Arch. Sup. for Prog. Lang. and Op. Sys.*, Mar. 2013.

[24] A. Ros, M. Davari, and S. Kaxiras, "Hierarchical private/shared classification: The key to simple and efficient coherence for clustered cache hierarchies," in *HPCA*, Feb. 2015.