Approximate Cache Coherence

Nandita Vijaykumar Amirali Boroumand
nandita@cmu.edu amirali@cmu.edu
Carnegie Mellon University

1. Introduction

Speeding up applications using multi-core/many-core architectures requires that applications be split into threads that execute concurrently on different cores. The challenge, however, is in handling how these threads communicate and access shared data. If programs are not correctly written without reasoned synchronization using primitives supplied by the ISA and hardware architecture, it could easily lead to data races and incorrect results because of non-deterministic interactions between threads. Mechanisms to ensure the correctness in such multi-threaded programs include coherence protocols in caches and hardware-enforced guarantees and primitives based on consistency models. These mechanisms ensure correctness but usually come at the cost of performance and are complex to implement. Synchronization between threads limit the scalability of an application by becoming serializing bottlenecks.

For example, coherence protocols are used to ensure that any data updates to shared data are propagated to all threads in the program. Typically Snoopy or Directory-based protocols are used to do this. An increase in the threaded-ness of a program would pressurize the protocol implementation in hardware leading to an increase in coherence misses in the caches and as well as overwhelming the interconnect and memory with messages and data transfers. At some point, any gains from parallel execution would be lost due to these overheads. A primary reason for synchronization complexity is the need to be assured of complete correctness and precision. But how important is precision in a program? Prior work [2] [3] have shown that there exist classes of applications that can, in fact, operate in modes of varying imprecision without catastrophic crashes. Examples of these applications include big data analytics, machine learning, augmented reality, image processing, etc. This leads to the possibility of trading off correctness for performance. This trade-off has been explored in the past to implement approximate computation or approximate storage [4].

The paper is organized as follows. Section 2 presents the background and motivation. Section 3 introduces our proposed approximate coherence protocol. Section 4 explains our methodology and Section 5 presents the results. Finally, Section 6 outlines our conclusion and future work.

Processor	
ISA	x86-64
CMP size and Core Freq.	4-core, OoO, 2 GHz
Re-order Buffer	128 entry
Cache Hierarchy	
L1 I-cache	64KB/2-way, 2-cycle
L1 D-cache	64KB/2-way, 2-cycle
L2 Cache	4MB/16-way, shared, 20-cycle
Coherence Protocol	MESI Directory
DRAM Parameters	
DRAM device parameters	Micron x8 4Gb chip, 8 chips
Memory Channel	DDR3 800 MHz channel, 1 channel
Number of banks	8 banks/device

Table 1: Simulator Parameter

2. Motivation

2.1. Coherence Traffic

Our goal in this project, hence is to reduce coherence traffic and coherence misses. As the first step, we did a study on PARSEC benchmark suite to measure coherence traffic. We used Gem5 full-system simulator [1] for this study. Gem5 is a modular event-driven full-system simulator which supports various instruction set architecture such as x86, ARM, Alpha, MIPS and PowerPC. We used x86 64-bit architecture for our evaluation mainly because this is the only architecture in Gem5 that fully supports various directory coherence protocols. The DRAM simulator is the DRAMSim which comes with Gem5 and provides both timing model and power model of DRAM. As for the operating system, we used Ubuntu 11.04 along with version 2.6.32.3 of the Linux kernel.

We used the Ruby memory model to monitor coherence traffic. Ruby implements a detailed model for the memory subsystem and enables modeling sophisticated cache coherence protocols. Our system configuration for this study is illustrated in Table 2. The interconnection network is a crossbar which means that each controller (L1/L2/Directory) is connected to every other controller via one switch. Packets, floating through network, are categorized as data or control packets. The control packet is 8 bytes and it includes control messages such as invalidations and write-back acks. The data packet is 64 bytes and it carries the data. We measure the traffic by monitoring packets flowing through each controller.

Figure 1 presents coherence traffic flowing through L1 cache controller. The y axis indicates the total number of bytes (data and control packets) normalized to the number of L1 cache misses. As it is shown in the figure 1, for most of the benchmarks, around 120 bytes traffic goes through each L1

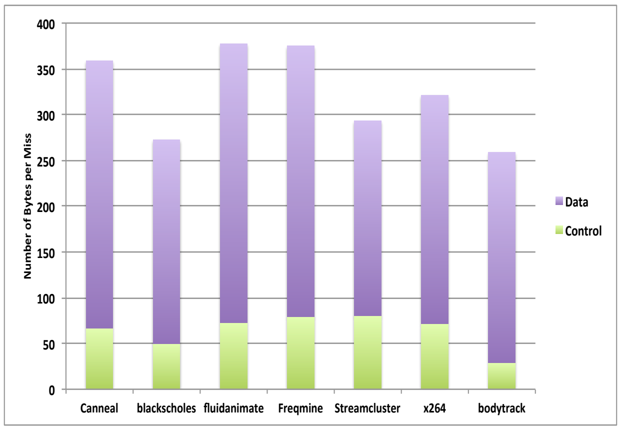

Figure 1: Number of bytes flowing through network per miss

cache controller per each L1 cache miss, instead of 64 bytes. This additional traffic, which is mainly dominated by data packets, is because of data exchange and coherence messages between threads sharing the same data. Figure 2 shows the coherence traffic across the network which is measured by monitoring packets going through all controllers (L1,L2 and Directory). For most of the benchmarks, the amount of traffic is around 300 bytes per cache miss. Our analysis indicates that when application exhibits high degree of sharing, coherence traffic becomes a major bottleneck, especially by increasing the number of threads.

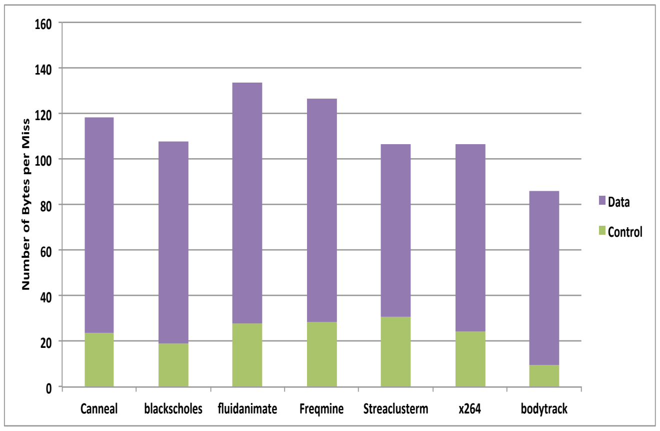

Figure 2: The number of bytes flowing through L1 cache per miss.

2.2. Approximate Coherence

Coherence ensures perfect correctness and precision of data. It guarantees that changes in shared data are propagated throughout the system in a timely fashion. The question is that is it always necessary? Is it always required to guarantee perfect correctness? This question leads us to the motivation of our project. If we could somehow tolerate imprecision, we would then be able to trade-off precision to reduce the coherence traffic. The next question would be what applications could tolerate such imprecision? As we mentiond, applications in approximate computing domain could fit well to this purpose. Let's walk through an example to see how we can exploit imprecision of data to reduce coherence traffic. Assume we have four threads, each reading and writing to a shared value, called X. Assume that X could be approximated and the error margin is 20%. Figure 3 shows a piece of code for such a program. It is assumed that the program is already annotated (data-centric approximtion). Figure 4 shows the cache content during the execution of this code. As it is illustrated in Figure 4a, thread 1 first reads the value X and brings it to its own cache (second line of code). At line 3, thread 1 performs a computation and updates the value of X. The approximated result of computation is 2.1 and the precise value is 2 (It is in the error margin). After that, it updates the value of X in the L1 cache (Figure 4b). The other threads then begin reading value of X (Figure 4c). At line 7, thread 2 executes the function. The approximated result of the function is 2.2 and the precise value is again 2. Thread 2 then wants to update the value of X, and to maintain cache coherence, it is required to obtain exclusive permission and invalidate all other shared copies. However, 2.2 is an approximate value of 2 and it is in the error margin. Since it is an approximated value of 2, both 2.2 and 2.1 could be treated as a same value. As a result, we could simply update thread 2 L1 cache and let other threads keep the previous value of X in their caches (Figure 4d). Such an approach allows us to eliminate 7 control messages (GetX request, Invalidations and Acks) and avoid coherence misses when other threads attempt to access value of X.

The above example indicates the key motivation behind our idea. Increasing number of thread makes coherence traffic a major bottlneck. It leads to excessive number of invalidations and coherence message, which lead to performance and power efficiency degradation.

```
1) @Approx<0.2> flout* x;
2) thread1.x = x;
3) x = thread1.function(200); // approximate function
4) Thread2.x = x;
5) Thread3.x = x;
6) Thread4.x = x;
7) x = thread2.function(250); // approximate function
```

Figure 3: Pseudo Code

3. The Proposal

In summary, our two primary goals in order to improve performance/energy efficiency are: • Reducing coherence misses • Reducing the amount of coherence traffic In the case of computing with approximable data, we can allow the usage of data that does not necessarily have the latest value. This would enable us to (i) drop writebacks and invalidates to memory and other cores during processor writes and (ii) execute on imprecise stale data in the case of a processor read. Ignoring all coherence messages for shared data would lead to a highly

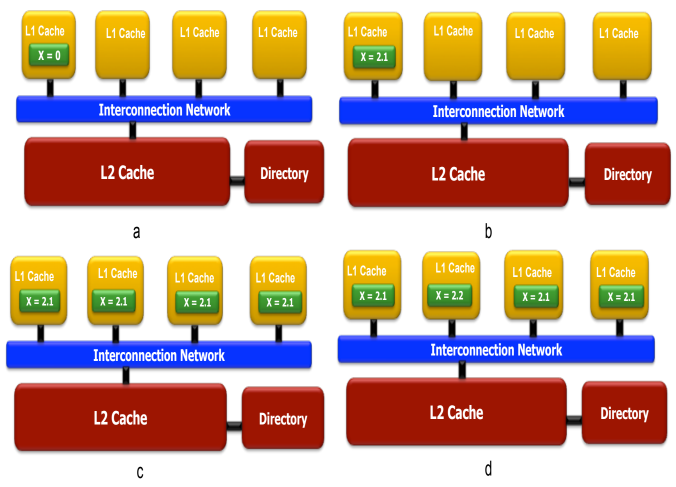

Figure 4: Cache content during code execution. (a) Thread 1 reads X (b) Thread 1 does the approximate computation and writes 2.1 to X. The precise value is 2. (c) Other threads read the value of x. (d) Thread 2 writes 2.2 to X. The precise value is again 2.

approximate and imprecise system. We, however, need to limit this imprecision by enabling coherence messages in such a manner so as to obtain an acceptable trade-off between precision and performance.

In this section, we describe our proposed approximate protocol. The key observation behind our proposed idea is that all updates are not equal. In fact, some writes make a huge difference while other slightly modify the previous data. As a result, we could treat them differently. It means that when there are only a small updates to the cache line, we might be able to avoid propagating those writes to other threads and eliminate the coherence messages plus avoid potential coherence misses.

The coherence traffic and coherence misses are mainly generated from transitions to the "Modified" state which results in invalidations to the same cachelines in other caches. In order to minimize these transi- tions, we propose to add another state to the cache coherence protocol to track modified data - "Highly Modified (H)" state. This state is treated in the same manner as a real modified state, where invalidates are sent to the directory and other caches and writebacks to DRAM are generated. The other modified state "Modified (M)" does not cause invalidations to data in the directory or other caches. In this case it is possible for a cacheline to exist simultaneously in the shared (S) and modified (M) states. The modified state primarily exists to control the transition from "H" and "M" and control impreci- sion. Upon eviction of data from the cache, data in the "M" state is still written back to memory as usual.

Figure 5 shows our proposed approximate cache coherence protocol. As it is shown in figure 5, when a big update reaches cache controller, it is treated as a regular write. It means that the cache line makes a transition to the H state and invalidate all other copies. Those copies could be in modified, shared or exclusive state. When a small updates gets to the cache controller, it simply makes a transition to the modified state.

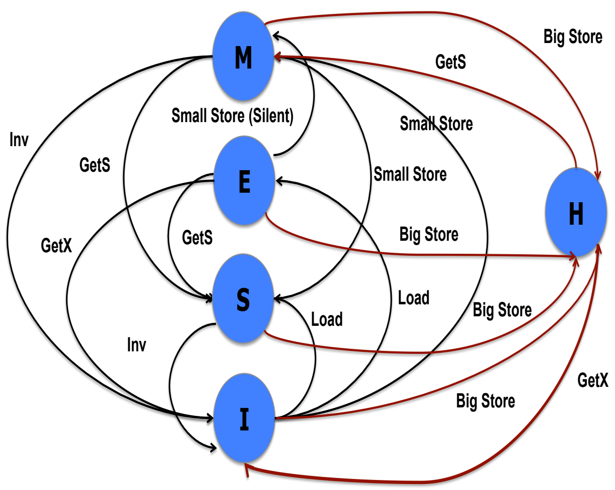

Figure 5: Approximate Coherence Protocol

4. Error Bounding

An important challenge with an approximate coherence protocol is error bounding. All forms of approximation at any level of the stack requires some mechanism to ensure that the imprecision introduced into the system doesn't cause catastrophic crashes to the system and provides a weak guarantee with respect to the error bound specified by the programmer. We discuss a few mechanisms to provide some sort of guarantee to the programmer. The error bounding mechanism here is essentially defining when an update is either "Big" or "Small".

4.1. Data-driven Error Bounding

Here, defining whether an update is "Big" or "Small" is defined by the size of the update in terms of the data value. If the size falls within a predetermined error threshold, it is treated as a "Small" update. Otherwise it is treated like a "Big" Update. The tricky part here is conveying the threshold from what is desired by the programmer to the hardware, so we can make this decision. Also we would need hardware to compare each data value during an update to determine how "Big" or "Small" it is. This can lead to high overhead.

4.2. Probability Error Bounding

We can also switch between "Big" and "Small" updates *probabilistically*. This has high overhead since we no longer consider the data value itself and seems use heuristics to determine the weight of the update. Some heuristics could include:

- Switch from "M" to "H" after a certain number of updates to a cache line.
- Choose from "M" and "H" based on the number of cores that are sharing the cache line
- Choose form "M" and "H" is a globally probabilistic way.

Cache Hierarchy	
L1 D-cache	64KB/4 way, 64B block size
Coherence Protocol	MESI Directory-based

Table 2: Simulator Parameters

5. Methodology

5.1. Infrastructure

We use PIN to perform our primary design analysis. PIN is dynamic binary instrumentation tool for x86 architecture that collects runtime information by inserting extra code into the program. PIN enables a much faster evaluation without the full architecutural simulation. Since Gem5 is too heavy-weight for initial experimentation, we built a cache model with flexible coherence support using PIN. We added the "H" state to our MESI-based cache simulator. We measure the difference in coherence misses, invalidate/write back traffic and directory lookups to determine the impact of the MESHI protocol.

5.2. Assumptions and Simplifications

We simply allow the approximate protocol to impact the cache performance for the workloads in our evaluation. Impacting the correctness and precision of the program as a result of our approximate cache protocol is beyond the scope of this project. Since we cannot measure the impact on precision of the program, we simply vary the *degree of approximation* as opposed to implementing any specific error bounding mechanism described above. We vary the degree of approximation by controlling the fraction of writes that are treated as "Big" as opposed to "Small" by the cache coherence protocol. The is entirely probabilistic and does not depend on the data within the cache line, nor the number of updates to any particular cache line. In that sense, the results below depict the worst case. We believe that more intelligent "Big" vs "Small" prediction mechanisms are likely to lead to better results. We, however, leave this evaluation as part of future work.

5.3. Workloads

We use a subset of benchmarks from the PARSEC-2.1 suite - namely *streamcluster, bodytrack, vips, fluidanimate, canneal*. We picked benchmarks that showed the highest amount of data sharing with short running times. We used the *simmedium* input set for our evaluation. We programmed our simulator to collect data within the Region of Interests within each benchmark and assume that all data touched within the ROI to be approximable.

6. Results

6.1. Coherence Misses

Figure 6 depicts the reduction in coherence misses with the MESHI protocol. The coherence misses include both: (1) Loads to data found in the *INV* state in the cache and (2) Stores to data found in the cache in a shared or invalid state. As expected, as the amount of approximation increases the coherence miss count reduces. What is interesting is that in most cases the drop in coherence misses is disproportionate with the fraction of "Big" writes. For example, treating 20% of all writes as "Big" reduces the number of coherence misses by close to 97% in most of the workloads that we evaluate here. The 25% point in *streamcluster* seems to be an anomaly here.

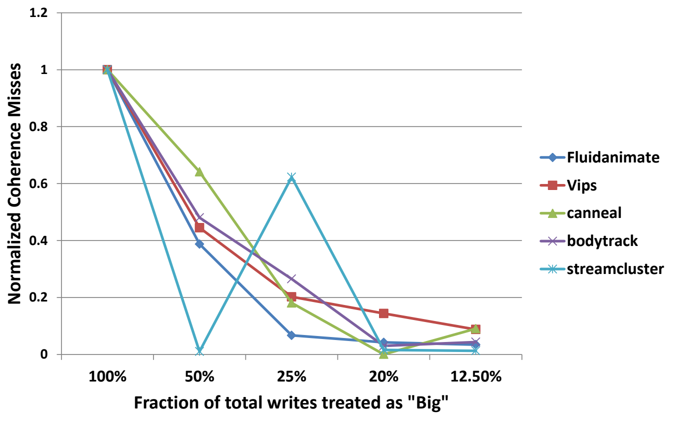

Figure 6: Reduction in Coherence Misses.

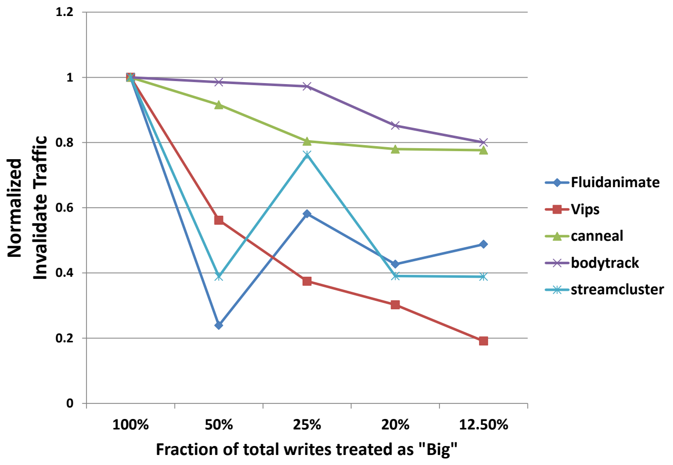

Figure 7: Reduction in Invalidate Traffic.

6.2. Invalidate Traffic

Figure 7 depicts the change in invalidate traffic as a result of the approximate coherence protocol. We define the number of invalidates the number of messages to other caches to invalidate cache lines because of the cache coherence protocol. We observe the impact of approximation varies across different workloads. For example, *bodytrack* benefits the least, with the reduction in invalidate traffic being only about 20% even with aggressive approximation. Whereas *vips* benefits a lot more commensurately with the approximation. We attribute this difference to the amount of sharing between different threads at any given time, where *vips* most likely has a lot more cores sharing a piece of data during an write operation than *bodytrack*.

6.3. Directory Lookups

Figure 8 depicts the number of the directory lookups as we vary the extent of approximation in the coherence protocol. Again we see a huge drop in the number of directory lookups

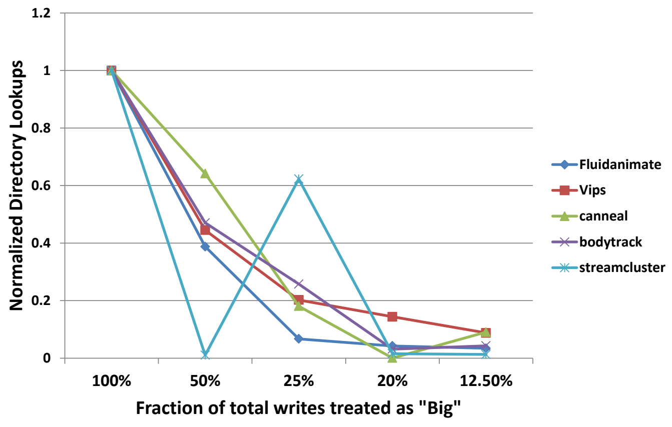

Figure 8: Reduction in Directory Lookups.

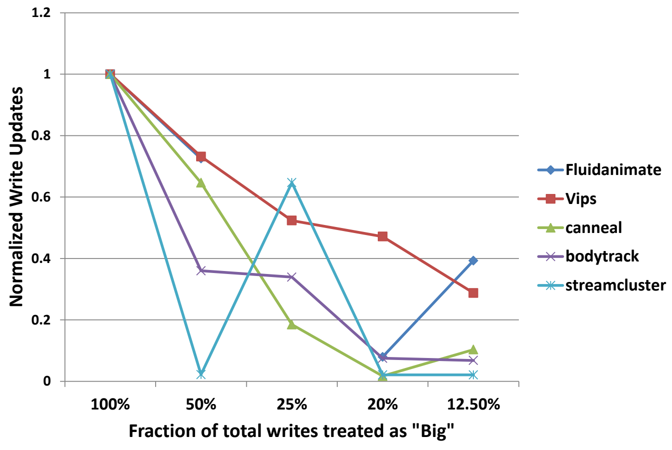

Figure 9: Reduction in Writebacks.

at a certain point (20%) in most of the workloads. Upto that point the reduction is more on-par with the fraction of "Big" writes. The "M" state allows for a lot more silent updates than before.

6.4. Writebacks

Figure 9 shows the number of cache lines that were shuttled back and forth across cores as a result of data sharing. As expected, approximation certainly provides a reduction in the number of cache lines that ping-pong between different cores. The extent again varies across the different workloads, with *Canneal* and *Bodytrack* benefitting the most.

7. Conclusion

From our evaluations of the impact of cache coherence in data-parallel workloads, we find that cache coherence adds a significant overhead in terms of latency, data-transmitted, energy and performance to the memory hierarchy. The amount of overhead depends on the extent of data communication between the threads. In this project, we trade-off precision to reduce this overhead of cache coherence. With this goal in mind, we proposed an approximate cache coherence protocol with flexible error control. Our evaluations show that an approximate coherence protocol reduces the number of coherence misses, directory lookups and invalidate traffic.

References

[1] N. Binkert, B. Beckman, A. Saidi, G. Black, and A. Basu, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, 2011.

[2] F. Niu, B. Recht, C. Ré, and S. J. Wright, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *In NIPS*, 2011.

[3] M. Rinard, H. Hoffmann, S. Sidiroglou, and S. Misailovic, "Patterns and statistical analysis for understanding reduced resource computing."

[4] A. Sampson, J. Nelson, K. Strauss, and L. Ceze, "Approximate storage in solid-state memories," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46. New York, NY, USA: ACM, 2013, pp. 25–36. Available: http://doi.acm.org/10.1145/2540708.2540712