Asymmetric Cache Coherency: Policy Modifications to Improve Multicore Performance

JOHN SHIELD, JEAN-PHILIPPE DIGUET, and GUY GOGNIAT,
Lab-STICC, Université de Bretagne-Sud

12

Asymmetric coherency is a new optimization method for coherency policies to support nonuniform workloads in multicore processors. Asymmetric coherency assists in load balancing a workload and this is applicable to SoC multicores where the applications are not evenly spread among the processors and customization of the coherency is possible. Asymmetric coherency is a policy change, and consequently our designs require little or no additional hardware over an existing system. We explore two different types of asymmetric coherency policies. Our bus-based asymmetric coherency policy, generated a 60% coherency cost reduction (reduction of latencies due to coherency messages) for nonshared data. Our directory-based asymmetric coherency policy, showed up to a 5.8% execution time improvement and up to a 22% improvement in average memory latency for the parallel benchmarks Sha, using a statically allocated asymmetry. Dynamically allocated asymmetry was found to generate further improvements in access latency, increasing the effectiveness of asymmetric coherency by up to 73.8% when compared to the static asymmetric solution.

Categories and Subject Descriptors: C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Adaptable architectures*

General Terms: Performance

Additional Key Words and Phrases: Nonuniform workload, cache, memory coherency, multicore processing, memory management

ACM Reference Format:
Shield, J., Diguet, J.-P., and Gogniat, G. 2012. Asymmetric cache coherency: Policy modifications to improve multicore performance. ACM Trans. Reconfig. Technol. Syst. 5, 3, Article 12 (October 2012), 12 pages.
DOI = 10.1145/2362374.2362376 http://doi.acm.org/10.1145/2362374.2362376

1. INTRODUCTION

FPGA multicore systems can allow for custom coherency policies to improve load balancing with methods that are complementary to task distribution and on-chip network arbitration. In multicore systems, the workload is not always evenly distributed. Sequential operations limit the parallel performance improvement as stated in Amdahl's Law and this limits the distribution of the workload by the designer and the operating system load balancing. We define the workload as a nonuniform workload (an asymmetric workload), when some cores are idle due to the sequential limitations of the workload. This situation is especially true for embedded multiprocessor architectures. Embedded MPSoC systems are often designed for specialized applications [Wolf et al. 2008], where they can accelerate the critical parts of the system. The possibility of accelerating a critical part of the workload is usually a sign of an asymmetric workload that can also benefit from asymmetric coherency.

Authors' address: J. Shield, J.-P. Diguet, and G. Gogniat, Lab-STICC, Université de Bretagne-Sud, F-56321 Lorient, France; email: {john.shield, jean-philippe.diguet, guy.gogniat}@univ-ubs.fr.

DOI 10.1145/2362374.2362376 http://doi.acm.org/10.1145/2362374.2362376

Memory coherency is necessary to ensure that all cores are using the most recent data values. However, the necessary drawback is there are overheads for maintaining the memory coherency. Coherency overheads are from the required additional coherency signals sent on the communication channel.

We present the new concept of asymmetric coherency, the purposeful design of the memory coherency to have different performances for different cores. Our work in asymmetric coherency addresses the nonuniform distribution of the application workload through modifications in memory coherency. The coherency modifications impact the communication latency and throughput for the memory system. We label the critical core (with the heavy workload) as the primary core and other cores (less important cores) to be secondary cores. We redistribute the coherency overheads to improve the memory latency and throughput of the primary core at the expense of secondary cores. This is approach to workload balancing accelerates parts of the workload and is complementary to workload distribution methods.

Asymmetric coherency is a policy change, so our design has low hardware cost and simplicity in implementation. The low cost and good performance shows that our technique is a suitable method for dealing with the asymmetry in multicore workloads.

Our theory of asymmetric coherency is a suitable customization for MPSoC systems. In this article, we present two asymmetric coherency policies as a proof of concept for the theory. Our previous work [Shield et al. 2011] was a bus-based coherency policy targeted at soft real-time systems or application specific systems, where the core requiring acceleration is known. We also extend our previous work by presenting a new directory-based coherency policy, analyzing multithreaded shared memory benchmarks, and using a full system simulator with operating system. The directory coherency is targeted for more general purpose systems MPSoC systems. The asymmetric settings are runtime adjustable and will only activate for shared data.

The remainder of this article is as follows: Section 2 describes related work; Section 3 explains the theoretical details behind the asymmetric coherency research; Section 4 describes the experimental tools and setup used; Section 5 presents the results that show asymmetric coherency improvements and describe discovered aspects of the behavior; Section 6 discusses where asymmetric coherency can be applied, based on our findings; and finally Section 7 concludes this article and mentions our future work.

2. RELATED WORK

Our previous work [Shield et al. 2011] presented results for a bus-based asymmetric coherency, for multiprogramming workloads that did not contain shared data. The simulation system was also dependant on recorded cache traces, which limited results to unshared data.

The new work in this article extends the research by considering multithreaded workloads that contain shared data. A directory-based system is considered for the asymmetric coherency, which is far more scalable than a bus-based system. The simulation system was changed to GEMS [Martin et al. 2005], which allows for full system simulation with an operating system. This allows us to demonstrate that our work can be complementary to operating system load balancing.

The closest related work in cache coherency has considered custom methods to address producer consumer or migratory data behavior [Bennett et al. 1990; Cheng et al. 2007; Cox and Fowler 1993; Martin et al. 2003; Stenström et al. 1993]. These optimizations work on the basis of creating special exceptions in the cache coherency to handle fine-grained behavior within an application. The previous work maintained the underlying cache coherency system and caters for special circumstances generated by the applications. The previous work required significant additional hardware to cater for these special circumstances and does not consider workload asymmetry.

The previous cache coherency research does not address the same problem as our work. Their aim is to improve coherency communications in general, whereas our coherency research aims to help solve workload balancing issues. Our proposed solution creates asymmetry in the performance of the cache coherency, which has not been considered before. Furthermore, unlike the previous work, our proposed asymmetric coherency policies are simple to implement.

There are several areas of research that address the problem of Amdahl's Law: processor frequency scaling [Annavaram et al. 2005]; heterogeneous processors [Becchi and Crowley 2006] containing different cache sizes/blocks/associativity; and quality of service (QoS) through bus arbitration [Iyer et al. 2007]. QoS of on-chip network arbitration is the most similar as it is concerned with communication. However, on-chip network arbitration only changes the priority of messages, while asymmetric coherency changes the quantities and inherent latencies of messages. All of this previous work is complementary as it can be used with asymmetric cache coherency.

An alternative solution to a shared memory coherency system, is to use private memories and message passing. However, a previous study [Chandra et al. 1994] found limited performance differences from using message passing over shared memory coherency. More recent design trends in reducing time to market in embedded systems emphasizes an additional drawback for message passing. Message passing requires that the software explicitly manages the shared data and this raises problems of increased design time for software development. However, message passing can be useful for increased system reliability [Kumar et al. 2011], when all the software is designed using message passing.

3. ASYMMETRIC COHERENCY DESIGN

The main theory behind our research concept is that an asymmetric performing cache coherency policy can help to improve the performance of asymmetric workloads by modifying the coherency overheads. In this section, we will present the details of our two asymmetric coherency policies.

3.1. Asymmetric Bus-Based Coherency

Our first presented asymmetric coherency policy is based on a bus-based system. This system will be used to demonstrate improvement for asymmetric coherency in multi-programming applications. Applications could either be a soft-real-time system that needs the critical code accelerated or an application specific system, where even distribution of the workload is not possible due to algorithmic limitations.

3.1.1. Implementation. The asymmetric coherency was developed from a MSI write-back/invalidate coherency and modified to provide writethrough/update operations for secondary cores. The modifications to the state machine are shown in Figure 1.

Behavior Description.

— The primary core does not need to check for modifications as it is always updated.
— The secondary cores always updates the primary core with any modifications.

Performance Differences.

— The primary core saves on coherency overhead, because it does not need to fetch the data modifications.
— Secondary cores take on an extra overhead in updating the primary core.

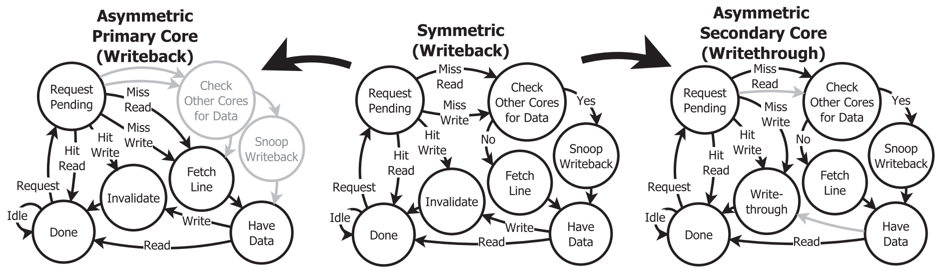


Fig. 1. Cache controller state machine changes to implement asymmetric coherency from an initial symmetric writeback (grey shows removed logic compared to the symmetric MSI writeback).

3.1.2. Hardware Costs and Scalability. The bus-based asymmetric coherency uses less hardware than the best performing symmetric system, which is writeback. The slight reduction is due to a simpler cache controller state machine. This asymmetric coherency is a trivial policy change that only modifies the cache controller state machine and does not require any additional hardware changes. The primary core is fixed at design time, so no interface hardware and no runtime adaptivity is required. A core only needs the reduced infrastructure for either asymmetric writeback or asymmetric writethrough, but not both.

Figure 1 shows the full extent of the hardware changes required. The implementation of the asymmetric coherency simplifies the original symmetric writeback system.

Bus interconnects are limited in scalability, so the bus-based coherency policy was designed with only one primary core. The one primary core limits the scalability of the design to smaller systems. However, this is an acceptable design limitation as the bus interconnect, which this particular coherency policy is based on, already limits the scaling of the system.

3.2. Asymmetric Directory-Based Coherency

Our second presented asymmetric coherency policy is designed for a nonbus interconnect (hierarchical switch), which requires a directory to keep track of coherency. We target general purpose MPSoC systems with this coherency policy. This system will be used to demonstrate improvement for asymmetric coherency in multithreaded applications.

3.2.1. Implementation. The directory-based asymmetric coherency was developed from a MOESI directory coherency. The modifications to the state machine from the standard MOESI directory coherency are shown in Figure 2.

The directory allows for tracking of cache lines that the primary core is sharing. Consequently, asymmetric operations can be restricted to only the shared cache lines of the primary core. The primary core sets the line to be volatile rather than invalidating the secondary cores on writes. Volatile causes secondary cores to read and write to the primary core directly. For all asymmetric cache lines, the secondary cores will always write to the primary core.

Behavior Description.

- The primary core has additional logic for choosing and setting special asymmetric states. The primary core sets the cache line to be volatile, instead of invalidating, which removes a wait for acknowledgement requirement.
- The secondary cores cater for two new asymmetric states. The asymmetric shared state always updates the primary core for writes, but performs reads locally. The

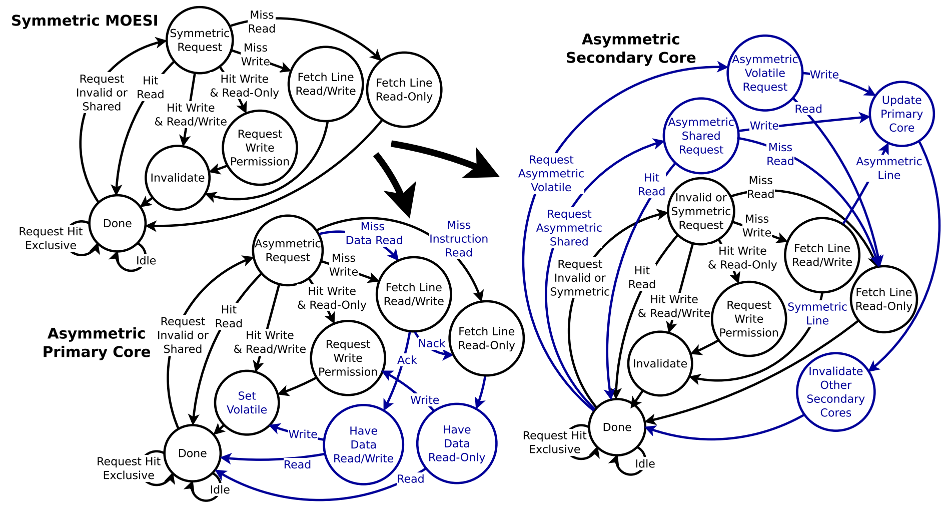


Fig. 2. Cache controller state machine changes to implement asymmetric coherency from an initial MOESI symmetric policy (blue shows added logic).

asymmetric volatile state denotes that the primary core has write permission (data is volatile), and both reads and writes are always directed at the primary core.

— The other cache line behavior is unchanged. All the nonshared cache lines and the cache lines shared between only the secondary cores behave in the same way as the original symmetric MOESI coherency policy.

Performance Differences.

— The volatile state removes the acknowledgement time for shared writes in the primary core.
— The additional asymmetric states mean that the primary core is never invalidated by the secondary cores. Consequently, it never needs to refetch invalidated cache lines.
— The secondary cores need to perform all the update functions for the shared cache lines of the primary core.

3.2.2. Hardware Costs and Scalability. To implement the asymmetric directory system, the following is required: a more complex state machine for the cache controllers (described in Figure 2), a single storage bit (to denote asymmetry) for each tag in the directory and the caches, and finally a processor accessible register to control setting of the primary core during runtime. The additional bit is to denote whether a line is an asymmetric cache line. In our experimental setup, the additional memory bit causes a 2% to 3% increase in the cache memory size. These hardware cost increases are trivial compared to the complexity and storage requirements of the original MOESI system.

Unlike the bus-based policy, a directory-based policy allows for tracking of exclusive cache lines and allows for normal symmetric coherency behavior between secondary cores. This significantly improves the scalability of the system (compared to the bus-based policy) as only data with known asymmetry will be targeted. If the asymmetry is targeted correctly to the data, our directory-based policy can always be configured to be equal or better than a standard coherency policy no matter what scaling of the system.

Furthermore, primary core settings for each individual cache line can ensure scalability of the performance improvement in much larger systems. Only a single primary core was used in the experiments, but a different primary core for each cache line is possible if a primary core bit is added for each cache tag. This provides very good scalability when used with good analysis for setting the asymmetry, because the asymmetry is handled on an individual cache line level.

4. EXPERIMENTAL SETUP

This section explains the two different experiment setups for the multiprogramming (previous research [Shield et al. 2011]) and multithreaded (new research) results. Two different simulation systems were used. The multiprogramming results use a faster simulator, which lacks the ability to model the operating system. The multithreaded results use a much slower full system simulator, which models the operating system behavior. Section 4.1 describes the simulation system used in the multiprogramming experiments. Section 4.2 describes the simulation system used in the multithreaded experiments.

4.1. Multiprogramming Experiment Setup for Asymmetric Bus-Based Coherency

MiBench [Guthaus et al. 2001] was used for the multiprogramming workload. Memory traces were recorded from the MiBench running on a Virtex 4 FPGA with Microblaze and uClinux.

A cycle accurate trace-driven simulation system [Shield et al. 2007] was upgraded to implement the bus-based asymmetric coherency (Section 3.1). The simulator uses memory traces for each core. The modified simulator has the additional ability to simulate a spinlock by simulating the polling of a memory location until the value matches the release signal. The simulator can load different configuration files to change the architecture and policies in the multicore system. It supports up to 16 cores. In the experiments, 1 primary core and 4 secondary cores were given workloads. The private caches used are 4 KBytes in size, 2 way associativity, Pseudo-LRU replacement and a 16 byte line size. The cores are connected to a DRAM main memory through a bus.

The experimental parameters modified were the system type, write policy and bus arbitration policy. Combinations of these parameters created the different systems tested. The details of the parameters are listed as follows:

- *System Type:* Single Processor; Multicore Symmetric Coherency; Multicore Asymmetric Coherency
- *Write Policy:* Writeback; WritebackOnHit; Writethrough
- *Bus Arbitration Policy:* Base (Round Robin); Priority (Bus Arbitration by Priority); Cancel (Bus Arbitration by Priority & Cancellation of Low Priority Accesses)

In the symmetric systems, all the cores use the same write policy parameter. In the asymmetric systems, the write policy mentioned only denotes the primary core policy. Writeback or writebackOnHit is used for the primary core and a writethrough policy is always used for the secondary cores.

4.2. Multithreaded Experiment Setup for Asymmetric Directory-Based Coherency

ParMiBench [Iqbal et al. 2010] was used for the multithreaded workload.

In the multithreaded experiments, the GEMS Simulator [Martin et al. 2005] was used to provide cycle-accurate full-system simulation results for the asymmetric directory coherency. The asymmetric directory coherency policy and MOESI coherency policy (Section 3.2) were implemented using the SLICC coherency description language in GEMS.

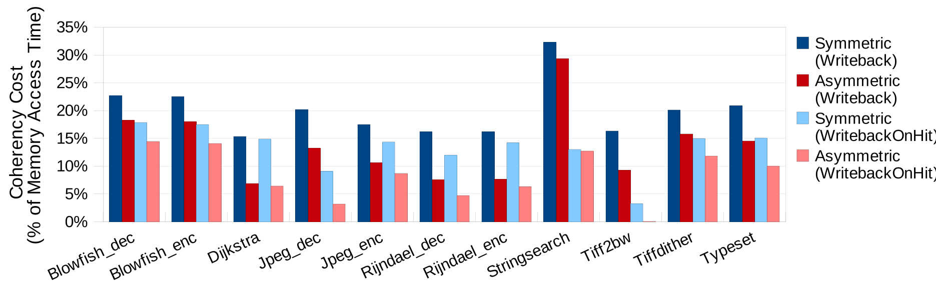


Fig. 3. Coherency costs of asymmetric and symmetric policies for single core workloads (MiBench applications) placed on a multicore system (Displaying percentage cost relative to memory access time).

The GEMS configuration settings were: hierarchical switch interconnect; Solaris 8 operating system; 4 UltraSPARC III cores; off-chip main memory; no L2 cache; and private 64 KBytes L1 caches with 4 ways. All other settings are default.

Two enhancements to the GEMS simulator were required to add timing delay in the state machines (to improve accuracy) and to allow write update operations.

Results for static and dynamic allocation of asymmetry were obtained. In the static solution, the asymmetry core is set for the entire execution of the application. In the dynamic solution, the best asymmetric core allocation is chosen periodically.

5. RESULTS

We present the performance results for our two asymmetric coherency policies using multiprogramming and multithreaded workloads.

Section 5.1 presents possible performance improvements for an asymmetric multiprogramming workload under asymmetric bus coherency. Section 5.2 describes the possible performance improvements for an asymmetric multithreaded workload under asymmetric directory coherency.

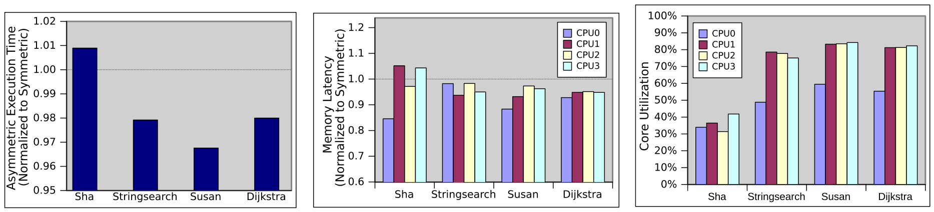
5.1. Asymmetric Multiprogramming Workloads

In multiprogramming workloads, some applications have a higher priority than others or sometimes soft real-time requirements. However, these applications often cannot run on more than a single core. Asymmetric policies can be used to accelerate a single primary core, while still allowing for secondary cores to continue running.

In this section, we demonstrate that a static setting of the bus-based asymmetric coherency can reduce overheads. This is only a small summary of our work with the bus-based asymmetric coherency. More detailed results for the bus coherency system can be found in previous research [Shield et al. 2011].

5.1.1. Asymmetric Coherency Improvement. The asymmetric coherency policy alleviates the cost of the coherency overhead. If the secondary cores are writethrough-always, checking their data is not required. This removes a bus extra cycle for checking whether there is dirty data in the secondary cores, when writeback is used in the secondary cores and the primary core needs to fetch data. This means that the asymmetric policy has an advantage over the symmetric policy when there is a read miss. Due to this difference the asymmetric policy reduced nonshared data coherency costs.

Figure 3 shows the coherency costs for the asymmetric and symmetric policies using both writeback and writebackOnHit under a wider range of MiBench applications. Applications that contained less than 2% system bus usage are not shown due to irrelevance as they do not access the main memory much.



(a) Execution Time (b) Memory Latency (c) CPU Utilization

Fig. 4. Asymmetric execution time, memory latency and CPU utilization for ParMiBench applications.

The difference between the coherency costs for the asymmetric policy and symmetric policy varied between 0.3% (for the application Stringsearch) and 8.7% (for the application Rijndael Decrypt). For the Stringsearch case, the coherency cost showed minor improvement with only a 2.3% and a 9.3% reduction in costs due to infrequent writes. However, in all the other cases the coherency costs were reduced by 20% to 60%.

5.2. Asymmetric Multithreaded Workloads

In multithreaded workloads, asymmetry in the cache coherency can improve performance by decreasing coherency costs for the sequential parts of algorithms, thus mitigating the effects of Amdahl's law. Decreasing the coherency costs of threads that access shared data more frequently than others can also accelerate the system.

In the multithreaded experiments, the asymmetric directory-coherency policy was used to test the impact of coherency on shared data. ParMiBench applications are used for the multithreaded workload.

Multiprogramming benchmarks running with the asymmetric directory policy showed no performance difference. The directory allows for tracking of shared cache lines, so asymmetry is designed to only be active for shared data. Consequently, asymmetric directory policy only has different behavior for shared data.

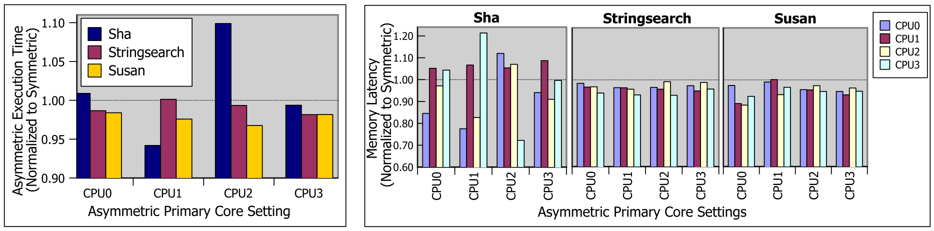
We first present static allocations of the primary core for the entire application execution. Then we present runtime allocation of the primary core.

5.2.1. Static Allocation of Asymmetric Coherency. In this section, our tests showed that workloads with uniform work distribution experienced a small performance improvement that does not come from asymmetry, instead deriving from better cache data retention with our policy. Workloads with nonuniform work distribution were able to achieve a larger improvement or reduction in performance, depending on the primary core setting. Consequently, it is important to set the primary core correctly for nonuniform workloads with shared data. However, uniform workloads can obtain a slight improvement without requiring any primary core adjustment.

An initial naive allocation of the primary core was conducted, based on setting the primary core to the parent thread on application startup.

Figure 4 shows the asymmetric coherency results normalized to the symmetric result for execution time and memory latency, and the CPU utilization of the applications.

Figure 4(c) gives the CPU utilization of each application, excluding the kernel usage that is mainly found on CPU0. The CPU utilization is fairly evenly distributed across the cores due to the operating system load balancing. Most of the applications effectively utilized the free CPU time on the cores, so they have a good uniform-workload distribution. However, the application Sha (Secure Hash Algorithm) shows that a significant amount of the time the CPUs are idle. The idle time indicates that



(a) Execution Time (b) Memory Latency

Fig. 5. Static allocation of the primary core for ParMiBench applications (normalized to symmetric results).

the application has been limited by its sequential operations, so the application Sha was a nonuniform workload.

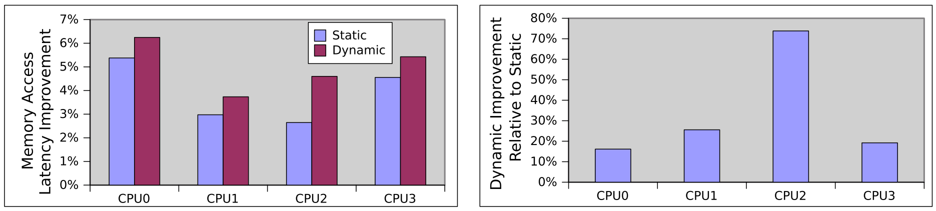
Figure 4(a) shows a few percent improvement for the uniform-workload ParMiBench applications, when using the asymmetric coherency. However, the memory latency for individual cores, in Figure 4(b), indicates the improvement is not caused by shifting coherency overheads to idle cores as the memory latency reduced for all the cores. Asymmetry in the coherency does not appear to impact uniform workloads.

Secondary cores show improvement in many cases. The improvement in the secondary cores can be explained by better retention of data within the caches. Detailed analysis of the secondary cores caches traces showed that the number of external memory misses decreased and misses that result in cache to cache transfers increased. Cache to cache transfers are far less costly than cache to external memory transfers, so more cache to cache transfers improves the performance. The decreased number of external memory misses is due to the reduced amount of invalidation required in the asymmetric coherency. Invalidation reduces the system caches to a single copy of the data, and this one copy can be flushed by the replacement policy before another core tries to access it. The asymmetric cache reduces invalidation and the larger number of copies reduces the likelihood for all copies of shared data to be flushed back to main memory when it is still needed.

The Sha application was a nonuniform workload, so setting the primary core is important as the primary impacts this type of workload. Further exploration of static primary core allocations was made with Sha, Stringsearch, and Susan, while the other applications were not explored due to the long simulation times required for full system simulation (Dijsktra took over 4 weeks to simulate once).

Figure 5 shows the results for the ParMiBench applications, using static allocations of the primary core of the asymmetric policy. For Sha, there was a 9.9% increase in execution time when the primary core was incorrectly allocated to CPU2 and a 5.8% decrease in execution time when the primary core was correctly allocated to CPU1. For Stringsearch, there was a 0% to 1.8% decrease in execution time depending on primary core allocation. For Susan, there was a 1.6% to 3.2% decrease in execution time depending on primary core allocation.

Figure 5(b) shows that 5.8% Sha execution time improvement was due to the redistribution of coherency overheads. The memory latencies of the four cores varied the most for the correct primary core setting of CPU1. Two cores exhibited higher memory latency with CPU1 as the primary core, while the other two experienced much lower latency. The lower latency for CPU0 and CPU2 (22.5% and 17.4% respectively) sped up sequential portions of the application and overcame the performance decreases found in CPU1 and CPU3 (6.7% and 21.3% decreases respectively). Unlike Sha, the Stringsearch and Susan applications were relatively unaffected by coherency asymmetry, because their workloads were relatively symmetric as shown in Figure 4(c).



(a) Memory latency improvement over symmetric policy

(b) Improvement of dynamic asymmetric over static asymmetric

Fig. 6. Memory latency for CPU cores over time for the application Sha.

Some improvement was still found in these two applications due to the impact of the better data retention in the caches, which was explained earlier.

5.2.2. Runtime Memory Latency Variation for Asymmetry. The number of shared accesses in an application varies significantly over time as the application performs different functions. Consequently, the improvement of asymmetric coherency can vary significantly over time too. Greater performance improvements in memory latency are possible by fine-tuning the coherency policy during execution time.

For the dynamic asymmetry experiment, only the first 6% of Sha's execution was run, due to the long simulation time of full system simulation. Dynamic allocation of the primary core was made every 1 million instructions, using an offline calculation.

Figure 6 shows the asymmetric policy memory-latency improvement for the application Sha using both static and dynamic allocations of the primary core. Overall the dynamic solution improves memory access latency between 16.2% to 73.8% compared to the static asymmetric solution. This shows further improvement is possible for asymmetric cache coherency if runtime adaptation is implemented rather than a static allocation for the entire application duration.

6. DISCUSSION OF FINDINGS

Asymmetric coherency was shown to be a method that redistributes coherency costs in a multicore processor system. Two coherency policies were discussed and different experiments with different workloads were conducted: a multiprogramming workload using a bus-based coherency system, and a multithreaded workload using a network on chip with directory-based coherency system.

Our bus-based asymmetric coherency showed that redistribution of coherency costs can be used to accelerate a single core at the cost of slow down in other cores. Due to the slowdown in other cores, this coherency policy is suitable only for customized MPSoC systems where the workload can be profiled and one of the cores is found to limit the performance of the entire system. This could either be a soft-real-time system that needs the critical code accelerated or an application specific system where even distribution of the workload is not possible due to algorithmic limitations.

Our directory-based asymmetric coherency targeted general purpose systems MPSoC systems. Our results show performance improvements for multithreaded applications when the application generates idle time on some of the cores in the system (asymmetry in the workload). The idle time can be due to either bad partitioning of the algorithm in the code, or partitioning limitations in the algorithm itself. The expected secondary core slowdown was instead a small improvement in many cases due to better retention of cache data. Consequently, for applications with little asymmetry in the workload, the results still showed some minor improvement due to the better cache data retention.

7. CONCLUSIONS

Our contribution is the concept of asymmetric coherency. The concept is to modify coherency policy to favor part of the workload where a sequential operations limits the performance of the system.

We have designed two examples, one bus-based asymmetric coherency policy and one directory-based asymmetric coherency policy. Using the bus-based system we have demonstrated that asymmetric coherency can reduce the inherent coherency costs of unshared data by 20% to 60% in multiprogramming applications. We used the directory-based system to look at the impact of shared data in multithreaded applications. We achieved an overall 5.8% execution time improvement for a parallel Sha application and memory latency reductions for some cores going up to 22%. Analysis of the application Sha shows that runtime adaptation of the coherency policy can further increase performance improvements. Our results, showing the benefits of runtime adaptation of coherency policy, motivates future work in implementing a decision algorithm for asymmetric-coherency runtime adaptivity.

REFERENCES

ANNAVARAM, M., GROCHOWSKI, E., AND SHEN, J. 2005. Mitigating Amdahl's Law through EPI throttling. *SIGARCH Comput. Archit. News 33*, 298–309.

BECCHI, M. AND CROWLEY, P. 2006. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *Proceedings of the 3rd Conference on Computing Frontiers*. ACM, New York, NY, 29–40.

BENNETT, J., CARTER, J., AND ZWAENEPOEL, W. 1990. Adaptive software cache management for distributed shared memory architectures. In *Proceedings of the 17th International Symposium on Computer Architecture*. 125–134.

CHANDRA, S., LARUS, J. R., AND ROGERS, A. 1994. Where is time spent in message-passing and shared-memory programs? In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*. 61–73.

CHENG, L., CARTER, J. B., AND DAI, D. 2007. An adaptive cache coherence protocol optimized for producer-consumer sharing. In *Proceedings of the IEEE 13th International Symposium on High Performance Computer Architecture*. 328–339.

COX, A. L. AND FOWLER, R. J. 1993. Adaptive cache coherency for detecting migratory shared data. In *Proceedings of the 20th International Symposium on Computer Architecture*. 98–108.

GUTHAUS, M. R., RINGENBERG, J. S., ERNST, D., AUSTIN, T. M., MUDGE, T., AND BROWN, R. B. 2001. MiBench: A free, commercially rep-resentative embedded benchmark suite. In *Proceedings in the IEEE International Workshop on Workload Characterization*. 3–14.

IQBAL, S., LIANG, Y., AND GRAHN, H. 2010. Parmibench - an open-source benchmark for embedded multiprocessor systems. *Comput. Archit. Lett. 9*, 2, 45–48.

IYER, R., ZHAO, L., GUO, F., ILLIKKAL, R., MAKINENI, S., NEWELL, D., SOLIHIN, Y., HSU, L., AND REINHARDT, S. 2007. QoS policies and architecture for cache/memory in CMP platforms. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. 25–36.

KUMAR, R., MATTSON, T. G., POKAM, G., AND VAN DER WIJNGAAR, R. 2011. A case for message passing for many-core computing. In *Multiprocessor System-on-Chip: Hardware Design and Tool Integration*. Springer.

MARTIN, M. M. K., HARPER, P. J., SORIN, D. J., HILL, M. D., AND WOOD, D. A. 2003. Using destination-set prediction to improve the latency/bandwidth tradeoff in shared-memory multiprocessors. In *Proceedings of the 30th International Symposium on Computer Architecture*. 206–217.

MARTIN, M. M. K., SORIN, D. J., BECKMANN, B. M., MARTY, M. R., XU, M., ALAMELDEEN, A. R., MOORE, K. E., HILL, M. D., AND WOOD, D. A. 2005. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News 33*, 92–99.

SHIELD, J., SUTTON, P., AND MACHANICK, P. 2007. Analysis of kernel effects on optimisation mismatch in cache reconfiguration. In *Proceedings of the 17th International Conference on Field Programmable Logic and Applications.* IEEE, 625–628.

SHIELD, J., DIGUET, J.-P., AND GOGNIAT, G. 2011. Asymmetric cache coherency: Improving multicore performance for non-uniform workloads. In *Proceedings of the 6th International Workshop on Reconfigurable Communication-Centric Systems-on-Chip*. 1–8.

STENSTRÖM, P., BRORSSON, M., AND SANDBERG, L. 1993. An adaptive cache coherence protocol optimized for migratory sharing. In *Proceedings of the 20th International Symposium on Computer Architecture*. 109–118.

WOLF, W., JERRAYA, A., AND MARTIN, G. 2008. Multiprocessor System-on-Chip (MPSoC) technology. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst. 27*, 10, 1701–1713.

Received August 2011; revised February 2012; accepted March 2012