Asymmetric Cache Coherency: Improving Multicore Performance for Non-uniform Workloads

John Shield, Jean-Philippe Diguet, Guy Gogniat
Lab-STICC, CNRS,
Université de Bretagne-Sud
Lorient, France
{john.shield, jean-philippe.diguet, guy.gogniat}@univ-ubs.fr

***Abstract*—Asymmetric coherency is a new concept to support non-uniform workloads in multicore processors. We present the theory behind asymmetric coherency policies and show our design requires no additional hardware over an existing system. Asymmetric coherency is designed to provide better performance for asymmetry in a workload and this is applicable to SoC multicores where the applications often are not evenly spread among the processors. The low cost and complexity makes it a desirable new coherency policy for future work. Our results show up to a 60% reduction in coherency costs for unshared data and up to a 174% improvement in memory access time for shared data.**

***Keywords*-Non-Uniform Workload; Cache; Memory Coherency; Multicore Processing; Memory Management**

I. Introduction

A key issue with multicore systems is how to allow applications to utilise the system effectively. Having a large number of cores usually creates the problem of distributing the workload optimally across the cores.

There are many cases where the workload placed on a multicore system is a non-uniform workload (an asymmetric workload), where some cores are often idle and others have a heavy workload. This is especially the case for embedded multiprocessor architectures. Embedded MPSoC systems are often designed for specialised applications [1], where they can accelerate critical parts of the system. This indicates that embedded applications commonly have non-uniform workloads that can benefit from asymmetric coherency.

Memory coherency has been cited as one critical area of research [2] that is needed for the envisioned massively parallel FPGA-based softcore multiprocessors to be successful. 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. The important coherency overheads are from the required additional coherency signals sent on the communication channel. There are inherent coherency overheads even when different applications do not share memory. Further coherency overheads are added when the applications do interact through shared memory.

Our work addresses the non-uniform distribution of the application workload through modifications in memory coherency. This impacts the communication latency and throughput for the memory system. We describe these distributions as asymmetric workloads or non-uniform workloads. We label the critical core (with the heavy workload) as the primary core and other cores (mostly idle cores) to be secondary cores. In this work, we propose a redistribution of coherency overheads to improve the memory latency and throughput of the primary core at the expense of secondary cores.

We present the new concept of asymmetric coherency, the purposeful design of the memory coherency to have different performance for different cores. Our design has low hardware cost, simplicity in implementation, and good performance. This shows that our technique is a suitable method for dealing with the asymmetry in multicore workloads.

Our theory of asymmetric coherency has immediate applicability in the design of non-runtime adjusted MPSoC systems. Furthermore, the presented research can be adapted for future implementations involving more complex coherency schemes and runtime adaptivity.

In this paper, we present an asymmetric coherency policy as a proof of concept for the theory. We are presenting the first model for asymmetric cache coherency and the first analysis on the behaviour of an asymmetric coherency system.

The remainder of this paper is as follows: Section II describes related work; Section III explains the theoretical details behind the asymmetric coherency research; Section IV describes the experimental tools and setup used; Section V presents the results that show asymmetric coherency improvements and describe discovered aspects of the behaviour; and finally Section VI concludes this paper and mentions our intended future work.

II. Related Work

The closest related work in cache coherency has considered custom methods to address producer consumer or migratory data behaviour [3], [4], [5], [6], [7]. These optimisations work on the basis of creating special exceptions in the cache coherency to handle fine-grained behaviour 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.

Our proposed solution instead considers creating asymmetry in the performance of the cache coherency, which has not been

considered before. Furthermore, unlike the previous work, our proposed asymmetric coherency policy is extremely simple to implement. There are no special circumstances in the policy or extra hardware to cater for the asymmetric coherency. Our implementation requires less hardware than the widely accepted writeback/invalidate coherency policy that has been in use for decades.

Related work in bus communication includes quality of service (QoS) of bus arbitration [8]. Bus arbitration is complementary to our work as our asymmetric coherency system combined with bus arbitration can improve overall performance to a greater degree than one mechanism can alone.

An alternative solution to a shared memory coherency system, is to use private memories and message passing. However, a previous study [9] found limited performance differences from using message passing over shared memory coherency. More recent design trends in reducing time to market in embedded systems emphasises an additional drawback for message passing. Message passing requires that the software has knowledge of the memory architecture and this raises problems of increased design complexity for software development.

III. Asymmetric Coherency Design

The main theory behind our research concept is that an asymmetric performing cache coherency policy can help improve the performance of asymmetric workloads by modifying the coherency overheads. In this section, we will explain why asymmetric workloads are relevant to many situations. Then we will present the details of how our asymmetric coherency modifies coherency overheads.

A. Asymmetric Workloads

Multicore systems allow for the workload to be spread among multiple cores. However, breaking the system workload over multiple cores does not mean the workload is evenly distributed among the cores.

Embedded MPSoC systems have specialised workloads that are often not evenly spread between the processors. In these kinds of cases, the asymmetry in the workload is algorithmic in nature.

Systems designed for general purpose workloads also contain asymmetry in the workload and sometimes deal with this by considering QoS. For the multiprogramming systems, workload asymmetry is caused by priority given to some programs while others are background tasks. For multi-thread systems, the workload asymmetry can be caused by either by the nature of the algorithm or the application designer not having the design time to evenly distribute the workload among cores.

In all these kinds of cases, the system can benefit from accelerating one of the processors at the cost of other processors. Our proposed solution is the acceleration of a primary core out of the multicore system through the use of asymmetric cache coherency. This will help the primary core to significantly run faster, by reducing coherency and data sharing costs, while still allowing for secondary cores to handle the more easily parallelisable workloads.

B. Design of Asymmetry Coherency Policies

This section will describe some key aspects of coherency, then explain the details behind asymmetric coherency.

The concept of asymmetric coherency can be applicable to many multiprocessor situations. However, as a starting point we focus on a small multicore system with a bus and broadcast snoop coherency. The simplicity of bus systems make them efficient for small multicore systems. A broadcast snoop coherency was used because it been one of the best solutions for small bus based multicores for the last decade. This example system lacks runtime configurability of the primary core, and can only handle one fixed primary core. All these restrictions can be removed with more design work.

1) Cache Coherency Background: Within a multicore, cache coherency is used to ensure that the latest written data is the data being used. There are two main operations for ensuring coherency: updating copies of the data, or invalidating copies of the data. There are also two methods of propagating the coherency operations: broadcasting the data to all cores, or providing a directory listing that records the status of data used by the cores.

The difference between update and invalidate is, update initially takes more time than the invalidate. However, the invalidate can incur further costs due to other cores requesting the updated data. The difference between the broadcast and directory systems is the broadcast must be received by every core, while the directory system can send data or invalidates to only the cores that require it.

Write policy is usually intimately linked to coherency. This work considers three write policies: writeback, writethrough and ”writeback only on hit/writethrough only on miss” (abbreviated writebackOnHit for this work). In bus systems, updates are usually performed with a writethrough and when a system decides to writeback it necessitates an invalidate. Consequently, writeback/invalidate and writethrough/update are common actions in coherency.

Techniques such as bus snooping, read-snarfing [10] and dual ported ram make broadcast superior for bus based systems. Snooping allows for writes from other cores to be captured, read-snarfing allows for capture of data fetches initiated by other cores, and dual ported rams allow processors to perform the snooping and read-snarfing tasks without pausing the operation of the processor cores. Dual ported rams removes a drawback, mentioned in older papers, where snooping would stop the processor from accessing its cache memory. Directory based systems are favoured for network on chip (NoC) due to scalability issues [4].

2) Asymmetric Coherency Design: The asymmetric coherency revolves around the idea of sacrificing the performance of some cores to improve the performance of other more important or greedier cores. It does this by changing the memory latency and bandwidth. Our example design,

TABLE I
THEORETICAL PERFORMANCE ADVANTAGE COMPARISON FOR THE SYMMETRIC AND ASYMMETRIC COHERENCY TYPES

Coherency Policy	Data Type	Comparison of Advantages Between Asymmetric & Symmetric Writeback/Invalidate
Symmetric Writeback/ Invalidate	Non-Sharing	**Secondary Core Write Hit:** Invalidate cost instead of a writethrough.
	Sharing	No differences.
Asymmetric	Non-Sharing	**Secondary Core Write Miss:** Single writes cost only a writethrough instead of a fetch.
	Sharing	**Primary Core Read and Write Misses:** Avoids checking other cores for data copies, and avoids writeback requests.
Coherency Policy	**Data Type**	**Comparison of Advantages Between Asymmetric & Symmetric Writethrough/Update**
Symmetric Write-through/ Update	Non-Sharing	**Primary Core Write Miss:** Single writes cost only a writethrough instead of a fetch.
	Sharing	**Primary Core All Writes:** No invalidation of secondary core copies. This reduces fetching. **Secondary Core Read Miss:** Avoids checking other cores for data copies, and avoids writeback requests.
Asymmetric	Non-Sharing	**Primary Core Write Hit:** Invalidate cost instead of a writethrough.
	Sharing	No differences.

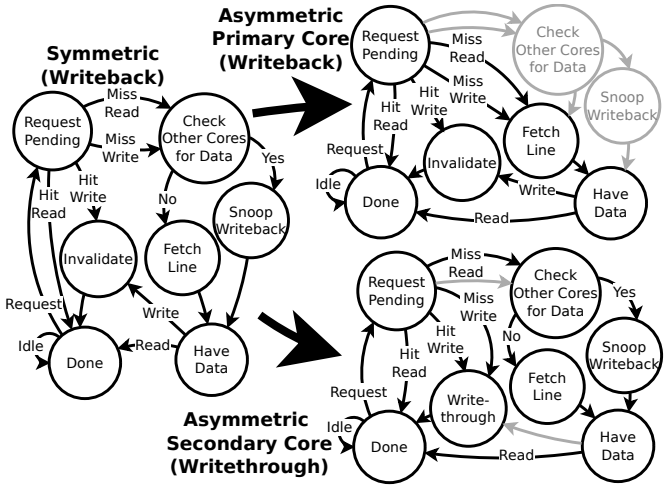

Fig. 1. Cache controller state machine changes to implement asymmetric coherency from an initial symmetric writeback (grey shows removed logic)

of the asymmetric coherency policy, aims to improve the performance of a primary core at the cost of secondary cores.

The improvement comes from combining the best aspects of the writethrough/update coherency policy with the write-back/invalidate coherency policy. With snooping and dual ported ram, the writethrough/update costs the initiating core more time in sending the writethrough instead of a shorter invalidate. The invalidate may save time for the initiating core, but it requires other cores to check for dirty data and request a writeback if any dirty data is found. If the primary core uses invalidates (cheaper for itself, more expensive for others) while the secondary cores use updates (cheaper for others, more expensive for itself), then the primary core should experience a performance improvement.

Table I shows the theoretical differences between our asymmetric policy and the two original symmetric policies.

3) Asymmetric Coherency Implementation and Cost: Our example asymmetric coherency uses less hardware than the best performing symmetric system, which is writeback.

The implementation of the asymmetric coherency simplifies the standard symmetric writeback system as shown in Fig. 1. The figure shows the full extent of the hardware changes required. The primary core uses a writeback/invalidate cache with a modification such it doesn't need to check for dirty data in other cores. The secondary cores use a writethrough/update cache with a modification to check for dirty data on other cores. An asymmetric coherency policy using a primary core write policy of writebackOnHit is also possible, but the system is similar enough that for brevity it will not be analysed extensively.

The slight reduction of hardware may not always be the case for more complex implementations of asymmetric coherency. Runtime configurability and many-core systems, opposed to multicore, will require more complex hardware.

General purpose systems will require runtime coherency configurability to handle both uniform and non-uniform workloads. Runtime coherency configurability imposes two more hardware costs that are: an extended memory controller state machine in each core to handle either writeback/invalidate or writethrough/update, and a system to handle the decision making. Many-core systems will likely require directory and hierarchical coherency. This may require additional status bits for data and more complex controllers.

IV. EXPERIMENTAL SETUP

This section will describe the experimental setup used to demonstrate the asymmetric coherency. Section IV-A describes the simulation system and Section IV-B describes the workloads used.

A. Simulation System

A cycle accurate and bit accurate trace driven simulation system [11], [12] was upgraded for this work. 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 4KBytes 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 below:

- **System Type:** Single Processor; Multicore Symmetric Coherency; Multicore Asymmetric Coherency
- **Write Policy:** Writeback; WritebackOnHit (Writeback Only on Hit/Writethrough Only on Miss); 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. Write-back or writebackOnHit is used for the primary core and a writethrough policy is always used for the secondary cores. If writethrough is used, our asymmetric coherency design becomes a symmetric writethrough system. Consequently, no asymmetric writethrough is described in our experiments.

Bus arbitration is complementary to asymmetric cache coherency so its side effects have been considered as part of this research. Our base system is round robin and we provide two priority policies to augment it. Firstly, a priority based system allows for a high priority core to be chosen before a lower priority core. Secondly, a more aggressive mode allows for a high priority core to interrupt and cancel a low priority memory access that has already started.

B. Benchmarks Setup

Asymmetric coherency can provide improvements for both multiprogramming and multi-threaded systems that are running asymmetric workloads. We will describe the experimental benchmark setups we use to show this.

In this section, we describe: the most relevant situations for multiprogramming and multi-threaded systems, and the benchmark and synthetic applications that were used for the experiments.

Initial analysis incorrectly suggests that multiprogramming applications can be run without coherency to avoid the overhead cost. However, this is not the case due to implementation issues. Coherency is just as relevant for multiprogramming applications, because they are run on general purpose systems that require coherency. The alternative to coherency is specialised partitioning that requires modifying each application to fit the memory architecture of the platform. Requiring many multiprogramming applications to be adapted to the memory architecture is extremely costly in design time.

The experiments outlined in this paper are designed to measure how the asymmetric coherency can change performance. Coherency costs are present even if data is not being shared. While not every memory access uses shared data, the infrastructure to ensure memory coherency applies an additional cost to most memory accesses.

There were several types of data sharing previously identified [3]. However, most adaptive coherency research [3], [4], [5], [6], [7] focuses on producer-consumer and migratory sharing as these types of data sharing encompass the majority of shared data. Producer-consumer sharing is data transferred once between cores. Migratory sharing is data that is transferred multiple times between cores.

For the multi-programming situation, applications often do not share large amounts of data between cores. Instead, the majority of overhead is likely to come from the inherent costs of the coherency infrastructure. Interaction between cores is mainly from cores competing for access to the bus. The multi-programming experiments in Section V-A will focus on how asymmetric coherency can reduce the inherent costs of coherency for unshared data.

For the multi-threaded situation, applications are likely to be processing data and sharing the results with other threads running on different cores. This fits the producer-consumer data sharing and migratory sharing models. The multi-threaded experiments in Section V-B will focus on how asymmetric coherency can reduce the costs of coherency for shared data. A producer-consumer test situation is used and the results are applicable to migratory sharing too. Unlike the previous adaptive coherency research, asymmetric coherency does not create specialised data transfers so migratory and producer-consumer sharing are similar.

A mixture of synthetic workloads and real applications are used in this research. Synthetic workloads are sometimes used to provide a fine grained control of memory accesses. These are randomly generated by our trace generator program using several parameters such as write percentage, data spread, time delay, address range, etc.

In the multi-programming examples, MiBench [13] will be used for the measured workload while synthetic benchmarks are used in other cores to simulate competing bus traffic. The MiBench traces were recorded from the MiBench running on a Virtex 4 FPGA with a Microblaze and uClinux. In the multi-threaded example, only synthetic workloads are used to allow for fine control of: the memory access patterns, the spinlocks, and the data sharing. The spinlocks synchronise the data transfers between cores.

V. Results

We describe several situations where the workload on a multicore system is asymmetric. Our results demonstrate that asymmetric coherency assists multi-programming workloads through better unshared data handling and assists multi-threaded workloads with better shared data handling.

Section V-A describes the possible performance improvements for an asymmetric multi-programming workload under asymmetric coherency. Section V-B describes the possible performance improvements for an asymmetric multi-threaded workload under asymmetric coherency.

A. Asymmetric Multi-programming Workloads

In multi-programming 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 allowing for secondary cores to continue running in the background.

In the multi-programming experiments, applications from the MiBench are used as the important workload. Secondary cores are modelled with synthetic workloads. The main interaction between cores is presented in terms of generated bus traffic. Data sharing is considered later for multi-threaded applications as most multi-programming applications will not share data.

In this section, we demonstrate the inherent overheads of coherency and how asymmetric coherency can reduce it. Then we explore the impact of bus traffic on the asymmetric coherency improvement. Finally, we describe the performance costs on secondary cores.

1) Basic Overheads for Multicore Memory Policies: Multicore processors require cache coherency to function, but maintaining the coherency incurs additional costs even when there is no data sharing. Asymmetric cache coherency can reduce the coherency costs. The asymmetric cache coherency policy uses a writethrough policy for the secondary cores. This reduces the inherent coherency overheads in the system for the primary core. In this section, we show that reducing these overheads can accelerate the primary core workload.

There are two overheads from supporting a multicore. Firstly, there is a bus arbitration cost, which in our simulated design is 2 clock cycles. Secondly, there is the cost for cache coherency checks that are present even if the other cores do not have a workload. Our system is a broadcast system without a central directory to keep track of sharing. Writes require invalidations of the data in other cores if they are not writethrough/update. Read misses must check for dirty data in other cores. Directory systems do not have the broadcast system overheads, but instead incur similar overheads from maintaining tags tracking the data sharing.

TABLE II
BREAKDOWN OF COHERENCY AND BUS ARBITRATION COSTS FOR UNSHARED DATA UNDER DIFFERENT CORE CONFIGURATIONS

		Memory Access Time Cost (%)	
Comparison to Single Core	**Primary Core Write Policy**	**Testcase Dijkstra**	**Testcase Jpeg Decode**
Shared Bus Cost	writeback	9.26%	7.48%
	writebackOnHit	9.89%	16.81%
	writethrough	16.30%	19.50%
Symmetric Coherency Cost	writeback	15.37%	20.22%
	writebackOnHit	14.90%	9.05%
Asymmetric Coherency Cost	writeback	6.89%	13.26%
	writebackOnHit	6.44%	3.19%

Table II shows the additional clock cycles required for coherency and bus arbitration costs as a percentage relative to the memory access time. Two examples are shown to explain coherency costs, the results for all the applications are in Fig. 2. Writethrough is not shown as it does not have an inherent coherency cost. However, the writethrough policy replaces the coherency costs with the cost of writethrough and consistently produced significantly worse results for all the real benchmarks tested throughout this work.

For the two MiBench examples in Table II, the bus arbitration costs varies between a 7.5% to a 19.5% increase in memory access time costs for symmetric and asymmetric systems. The symmetric coherency policy cost with writeback varies between 9.1% to 20.2%. The asymmetric coherency policy cost varies between 3.2% to 13.3%.

2) Asymmetric Coherency Improvement: The asymmetric coherency policy alleviates the cost of the coherency overhead. When writeback is used in the secondary cores, and the primary core needs to fetch data, a bus extra cycle is needed for checking whether there is dirty data in the secondary cores. If the secondary cores are writethrough always the check for dirty data is not required. 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 compared to the symmetric policy reduced non-shared data coherency costs by around 5.9% to 6.9% in Table II.

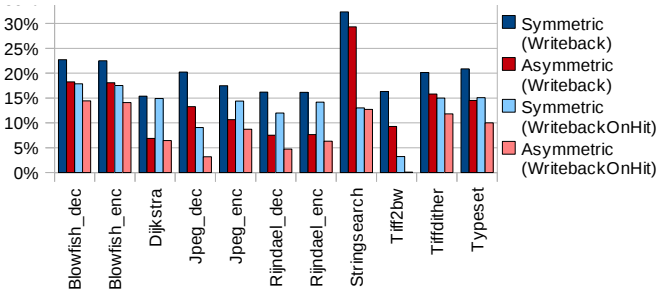

Fig. 2. Coherency costs of asymmetric and symmetric policies for single core workloads (MiBench applications) placed on a multicore system (Displaying percentage cost relative to memory access time)

Fig. 2 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.

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% in Fig. 2.

3) Secondary Core Traffic Impact on the Workload: Even though the other cores do not handle any of the measured workload, they can influence the operation of asymmetric and symmetric policies. The secondary cores influence the asymmetric and symmetric policies through data sharing and competing bus traffic. In a multiprogramming environment, the impact of the secondary cores on the primary core is mainly caused by the differences in bus traffic generated by different policies. Data sharing is less common in a multiprogrammed environment and will be discussed in the multi-threaded section.

The amount of bus load from the secondary cores is dependant on the type and the frequency of memory accesses performed by the secondary processors. This section will look at the impact from read and write accesses generated by the secondary cores.

For this experiment, only the Dijkstra application is shown as other MiBench applications exhibited similar behaviour. Synthetic secondary core workloads are used to allow for adjustment of secondary core memory access patterns.

The results shown in Table III demonstrate secondary core

TABLE III
IMPACT OF SECONDARY CORE READ/WRITE TRAFFIC ON ASYMMETRIC POLICIES WITH DIJKSTRA ON A WRITEBACK PRIMARY CORE (MEMORY ACCESS TIME NORMALISED TO SINGLE CORE WITH NO TRAFFIC)

Secondary Core Traffic	Bus Arbitration Policies	Symmetric Coherency		Asymmetric Coherency		Asymmetric Coherency (Memory Access Time Improve %)
		Access Time	Bus Usage (%)	Access Time	Bus Usage (%)	
High Read Traffic	Base	7.83	100.0	6.69	100.0	17.07
	Priority	2.37	100.0	2.04	100.0	16.33
	Cancel & Priority	1.30	100.0	1.21	100.0	7.78
Med Read Traffic	Base	3.36	84.7	2.69	79.7	25.09
	Priority	2.15	85.3	1.86	80.3	15.95
	Cancel & Priority	1.30	95.6	1.21	91.4	7.79
Low Read Traffic	Base	1.53	41.4	1.37	37.9	11.43
	Priority	1.50	41.6	1.34	38.1	11.32
	Cancel & Priority	1.28	51.0	1.19	46.3	8.10
High Write Traffic	Base	1.39	95.9	3.31	100.0	-57.86
	Priority	1.26	95.8	1.48	100.0	-14.57
	Cancel & Priority	1.26	95.8	1.21	100.0	4.13
Med Write Traffic	Base	1.27	31.6	1.70	66.8	-25.05
	Priority	1.26	31.7	1.40	67.1	-9.52
	Cancel & Priority	1.26	31.8	1.20	67.0	5.08
Low Write Traffic	Base	1.27	27.0	1.25	33.5	1.11
	Priority	1.26	27.0	1.23	33.6	2.74
	Cancel & Priority	1.26	27.1	1.18	33.5	6.67
None	All Types	1.26	25.6	1.17	23.6	7.96

read and write access traffic impacting the primary core running the Dijkstra application. The secondary core workloads are considered background processes, so the results focus on the impact on the primary core.

Under secondary core read traffic, the asymmetric coherency policy was superior due to requiring fewer bus accesses for coherency. This advantage peaked at a 25% improvement with moderate bus traffic. Under secondary core write traffic, the asymmetric coherency policy generated a higher bus traffic than the symmetric. This is due to the writethrough policy of the secondary cores in the asymmetric coherency policy. The policy increased bus traffic and caused significant reductions in performance for the asymmetric policy. In the worst case situation, there was a 58% decrease.

However, when priority bus arbitration policies were applied, the impact of bus traffic was less important. For read traffic the performance improvement was between 7.8% to 8.1% and for write traffic it was between 4.1% to 6.7%. The single workload coherency improvement without bus traffic was 8.0%. Secondary core workloads can cause a significant traffic related performance detriment to the primary core, for both the asymmetric and symmetric coherency policies. However, the priority bus arbitration policies removed most of these performance detriments.

4) Secondary Core Performance Degradation from Asymmetric Coherency Policies: The asymmetric policy is designed to sacrifice performance in the secondary cores to improve the performance of the primary core. In a multiprogramming environment the cost will be the performance of the background processes running on the secondary cores.

The results in Table IV showed that for asymmetric coherency the maximum performance decrease was solely due to the secondary core writethrough policy. The maximum cost of using writethrough was 31% and this decreased as bus traffic was added. When the bus saturated, the asymmetric coherency gave an improvement over the symmetric. This is because increasing bus traffic, combined with round robin arbitration,

TABLE IV
IMPACT OF ASYMMETRIC POLICIES ON PERFORMANCE OF ONE SECONDARY CORE RUNNING DIJKSTRA (NORMALISED TO A WRITEBACK SYMMETRIC SYSTEM WITH NO BACKGROUND TRAFFIC)

Traffic Type	Bus Arbitration Policies	Symmetric Coherency		Asymmetric Coherency		Asymmetric Coherency (Secondary Core Memory Access Time Improve %)
		Memory Access Time	Bus Usage (%)	Memory Access Time	Bus Usage (%)	
High Traffic Primary Core	Base	3.72	98.3	3.83	99.8	-2.88
	Priority	5.38	98.6	7.04	100.0	-23.59
	Cancel & Priority	441.16	100.0	356.39	100.0	23.79
High Traffic Secondary Core	Base	4.41	99.8	4.01	100.0	9.93
	Priority	4.46	99.8	4.24	100.0	5.18
	Cancel & Priority	4.37	99.9	4.35	100.0	0.45
Med Traffic Secondary Core	Base	2.13	71.8	2.56	83.1	-16.83
	Priority	2.26	71.5	2.74	83.1	-17.48
	Cancel & Priority	2.58	76.4	3.05	86.1	-15.51
Low Traffic Secondary Core	Base	1.36	49.0	1.92	58.4	-28.87
	Priority	1.37	49.0	1.93	58.3	-29.07
	Cancel & Priority	1.73	52.2	2.26	60.3	-23.54
No Traffic	All Types	1.00	25.7	1.45	36.3	-31.01

favours the fewer bus accesses of the writethrough coherency, opposed to the shorter overall access time of the writeback coherency. However, when the primary core saturates the bus, the cancel bus arbitration caused the bandwidth starvation of the secondary cores.

B. Asymmetric Multi-threaded Workloads

This section looks at the possible performance improvement asymmetric coherency policy can provide for a multi-threaded application, where the workload is also asymmetric.

The test cases model a producer/consumer situation, consisting of a single primary consumer thread with four data producer threads that feed the primary thread data. Each thread is associated with a core. The primary core load is much higher than the individual secondary core loads. Spinlocks are used to ensure the cores exchange data correctly, and our memory access time results in this section will include the time the cores spend idling in spinlocks.

There are three considerations that affect the performance of the asymmetric coherency under multi-threaded workloads. These are in addition to the behaviour discussed in Section V-A. The considerations are: the memory access pattern of the data sharing, the percentage split of shared versus non-shared data, and the bus usage of the multi-threaded workload.

This section will first explore the data sharing patterns and the percentage split of shared versus non-shared data for asymmetric policies. Then it will explore the impact of bus usage on the multi-threaded performance of the asymmetric policies.

1) Sharing Interactions in a Multi-threaded System: We first look at the impact of data sharing patterns between the cores and the percentage of shared versus non-shared data.

The possible read and write interactions for data sharing between cores are labelled in Table V. The basic producer/consumer example was adapted with different types of interactions and with varying percentages of non-sharing writes performed by the primary core.

The main advantages of a symmetric writethrough policy are that it avoids data sharing overheads in the coherency and unnecessary fetches for writes. The main advantage of

TABLE V
TEST CASE LABELLING FOR POSSIBLE DATA SHARING INTERACTIONS BETWEEN SECONDARY AND PRIMARY CORES

		Secondary Core		
		Read	Write	Read/Write
Primary Core	Read	p.r_s.r	p.r_s.w	p.r_s.rw
	Write	p.w_s.r	p.w_s.w	p.w_s.rw
	Read/ Write	p.rw_s.r	p.rw_s.w	p.rw_s.rw

a symmetric writeback policy is it reduces unnecessary writes to the main memory for unshared data.

The proposed asymmetric policy combines the advantages of the two symmetric policies for the operation of the primary core. From the perspective of the primary core, the asymmetric policy has the performance of the writeback policy for the unshared data while operating with the reduced data sharing coherency overheads of the writethrough policy.

Fig. 3 shows producer consumer data sharing, where the primary core is the consumer and four secondary cores are producers of the data. The secondary cores process data packets before passing the data to the primary core with a spinlock. The types of data sharing explored in Fig. 3 were labelled in Table V. The results show the increased memory access latency of the best symmetric configuration relative to the best asymmetric configuration. The memory access time improvement ranged from -3.5% to 174% and this corresponded to an execution time improvement of -1.3% to 63.7%.

There were nine test cases explored in Fig. 3. In three of the test cases, the data producers (secondary cores) do not produce any data, which does not comply with the producer consumer situation. The three test cases p.r_s.r, p.w_s.r and p.rw_s.r were included only to help explain coherency behaviour. In these cases, there is no improvements observed for the asymmetric policy as the secondary cores only performed reads. Coherency deals with the propagation of write data, so no actual coherency related data sharing is occurring.

The other six cases, where the secondary cores writes to the shared data, show improvement from the asymmetric coherency policy. Most of these cases show zero improvement for the asymmetric policy at 100% data sharing and at 0% data sharing. However, there is significant improvement around 70% data sharing.

This effect can be explained by considering that the results are composed of the individual symmetric policies as shown for p.rw_s.w in Fig. 4. For 100% shared data, writethrough in the secondary core reduces the costs for the primary core. When there is 0% shared data (100% unshared data), the writeback policy provides the best efficiency for the primary core. However, the symmetric policies are best for either the shared or the unshared data, but not both.

The asymmetric policy provides the advantages of both symmetric policies for the primary core. Consequently, when there is a mixture of shared and unshared data asymmetric coherency is superior. The decreased efficiency of data accesses on the secondary cores did not effect the primary core,

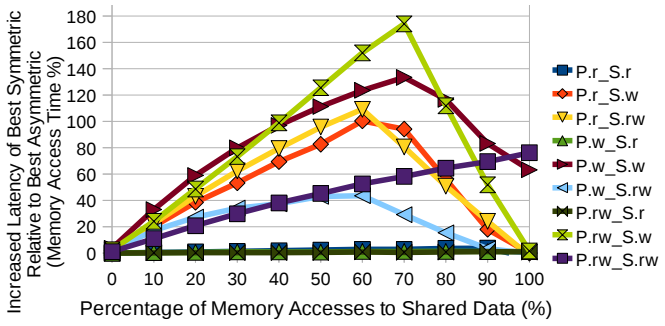

Fig. 3. Increased access time latency of symmetric compared to asymmetric coherency. Sharing interaction labels are explained in Table V

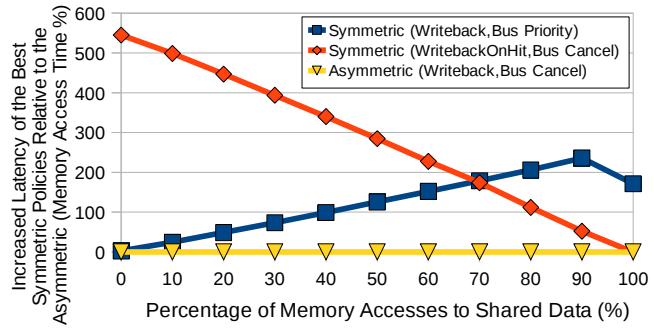

Fig. 4. The best system results that were used to calculate the sharing performance for the data sharing p.rw_s.w in Fig. 3

because of the asymmetric workload. The primary core had a larger workload, which meant that the secondary cores were still able to prepare their data for the primary core before the primary core required it.

In most cases, symmetric writethrough and writebackOnHit were able to equal the asymmetric policy at 100% data sharing. However, there are two cases, p.w_s.w and p.rw_s.rw, where the asymmetric policy is better at 100% data sharing. In p.w_s.w, the asymmetric was better than the symmetric writethrough and writebackOnHit due to being able to cache the writes on the primary core without coherency costs. In p.rw_s.rw, the symmetric writethrough and WritebackOnHit are worse than the asymmetric for differing reasons. The symmetric writethrough is worse due to not being able to cache the primary core writes. While the symmetric WritebackOnHit is worse due to the writeback on the secondary core requiring a writeback request from the primary core.

Asymmetric outperforms all of the symmetric coherency policies when there is a mix of sharing and non-sharing data accesses. The memory access time improvement was -3.5% to 174%, and there was a corresponding execution time improvement of -1.3% to 63.7%. Furthermore, when comparing individual policies, a single symmetric policy will perform poorly at either 100% data sharing or 0% data sharing. However, a single asymmetric policy provides good performance in all the situations.

2) Bus Usage Impact on a Multi-thread System: All the bus usage of a multi-threaded system is part of the workload, so

TABLE VI
IMPACT OF THE MULTI-THREADED WORKLOAD BUS USAGE ON THE MEMORY ACCESS TIME FOR ASYMMETRIC COHERENCY (BEST RESULT HIGHLIGHTED FOR THE SYMMETRIC AND ASYMMETRIC SYSTEMS)

			Low Traffic		Med Traffic		High Traffic		Highest Traffic	
Description	Write Policy	Bus Policy	Norm. Access Time	System Bus Usage (%)	Norm. Access Time	System Bus Usage (%)	Norm. Access Time	System Bus Usage (%)	Norm. Access Time	System Bus Usage (%)
Symmetric Multi-core	Writeback	Base	2.250	26.53	2.676	40.61	2.816	51.13	1.260	77.77
	WritebackOnHit	Base	2.570	29.55	2.932	45.77	3.066	57.63	1.706	89.18
	Writethrough	Base	3.099	38.41	3.428	58.64	3.583	72.56	1.974	92.43
Symmetric Multi-core with Priority Bus Arbitration	Writeback	Priority	2.163	26.79	2.622	40.96	2.786	51.44	1.234	79.18
	Writeback	Cancel	1.835	29.50	2.208	81.80	3.445	92.17	1.141	97.82
	WritebackOnHit	Priority	2.469	29.90	2.761	47.01	2.951	58.88	1.697	89.63
	WritebackOnHit	Cancel	1.937	30.50	2.224	69.03	2.741	81.69	1.573	97.96
	Writethrough	Priority	2.937	39.10	3.257	60.12	3.470	73.97	1.964	92.84
	Writethrough	Cancel	2.934	38.67	3.219	75.02	3.660	86.02	1.836	95.22
Asymmetric Multi-core	Writeback	Base	1.660	19.51	1.835	32.03	1.907	42.40	1.145	79.86
	Writeback	Priority	1.541	19.81	1.640	33.26	1.770	43.83	1.126	81.03
	Writeback	Cancel	1.000	20.12	1.000	61.21	1.000	92.72	1.000	96.97

the impact is different from interference bus traffic presented in Section V-A3. In this section, the bus usage is increased or decreased by adjusting the workload's pseudo-processing time between memory accesses.

Table VI shows the effect of increasing the bus usage for a test case equalivant to the p.rw_s.w at 70% sharing. These test conditions showed high improvement in the previous section. To generate low bus usage a long delay was used between memory accesses, while the high bus usage was generated with shorter time delays.

The results showed that asymmetric coherency is better suited to improving performance of the primary core during increased bus usage until the bus reaches a saturation point. When the bus usage saturates the performance drops significantly. This is due to the cancel bus arbitration policy causing starvation in the secondary cores while the primary core is completing its own workload (as seen in Section V-A4). When this occurs, in a multi-threaded application, the primary core and secondary cores are not working in parallel.

VI. CONCLUSIONS

Our contribution is the concept of asymmetric coherency. We have designed an example and shown possible benefits.

Asymmetric coherency is a promising new concept for the design of memory coherency. It provides another method that hardware can be fine tuned to the asymmetry found in many workloads. Our example design has an extremely low cost and complexity. This makes the concept immediately applicable to embedded MPSoC systems.

In the multiprogramming experiments, our results show the asymmetric policy reduces coherency costs by 20% to 60% for most of the test cases. We also measured the impact of secondary core bus traffic. Traffic generated by reads would increase the asymmetric policy improvement, while traffic generated by writes would decrease the asymmetric policy improvement. However, with priority bus arbitration policies, the impact from considering secondary core traffic was very small.

Our multi-threaded experiments focused on shared data handling with the primary core dependant on secondary core data. The results showed improvements in execution time of up to 63.7% for the asymmetric compared to the symmetric coherency. This corresponded to a 174% improvement in memory access time. The asymmetric coherency gave an improvement in almost all of the data sharing situations.

Future work will look at implementing runtime adaptivity for cache coherency policies. Adaptivity will make the asymmetric coherency policy useful in general purpose systems. The runtime adaptivity will allow for configurability in the choice of the primary core and also coherency policy changes between symmetric and asymmetric configurations. Furthermore, we intend to create asymmetric coherency designs that can scale to many-core architectures. This involves designing for different memory hierarchy and interconnect situations.

REFERENCES

[1] W. Wolf, A. Jerraya, and G. Martin, "Multiprocessor system-on-chip (mpsoc) technology," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 10, pp. 1701–1713, 2008.

[2] T. Dorta, J. Jimenez, J. Martin, U. Bidarte, and A. Astarloa, "Overview of fpga-based multiprocessor systems," in *International Conference on Reconfigurable Computing and FPGAs*, Cancun, Mexico, 2009, pp. 273–278.

[3] J. Bennett, J. Carter, and W. Zwaenepoel, "Adaptive software cache management for distributed shared memory architectures," in *Proceedings of the 17th International Symposium on Computer Architecture*, Seattle, WA, USA, 1990, pp. 125–134.

[4] A. L. Cox and R. J. Fowler, "Adaptive cache coherency for detecting migratory shared data," in *Proceedings of the 20th International Symposium on Computer Architecture*, San Diego, CA, USA, 1993, pp. 98–108.

[5] P. Stenström, M. Brorsson, and L. Sandberg, "An adaptive cache coherence protocol optimized for migratory sharing," in *Proceedings of the 20th International Symposium on Computer Architecture*, San Diego, CA, USA, 1993, pp. 109–118.

[6] L. Cheng, J. B. Carter, and D. Dai, "An adaptive cache coherence protocol optimized for producer-consumer sharing," in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, Phoenix, Arizona, USA, 2007, pp. 328–339.

[7] M. M. K. Martin, P. J. Harper, D. J. Sorin, M. D. Hill, and D. A. Wood, "Using destination-set prediction to improve the latency/bandwidth tradeoff in shared-memory multiprocessors," in *Proceedings of the 30th International Symposium on Computer Architecture*, San Diego, CA, USA, 2003, pp. 206–217.

[8] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt, "Qos policies and architecture for cache/memory in cmp platforms," in *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, San Diego, CA, USA, 2007, pp. 25–36.

[9] S. Chandra, J. R. Larus, and A. Rogers, "Where is time spent in message-passing and shared-memory programs?" in *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, 1994, pp. 61–73.

[10] C. Anderson and J.-L. Baer, "Two techniques for improving performance on bus-based multiprocessors," in *Proceedings of the 1st IEEE Symposium on High-Performance Computer Architecture*, Raleigh, NC, USA, 1995, pp. 264–275.

[11] J. Shield, P. Sutton, and P. Machanick, "Analysis of kernel effects on optimisation mismatch in cache reconfiguration," in *Proceedings of the 17th International Conference on Field Programmable Logic and Applications*. Amsterdam, The Netherlands: IEEE, 2007, pp. 625–628.

[12] J. Shield, "Dynamic cache switching: Developing configurable caches for use in softcore processors," Ph.D. dissertation, The Univerisity of Queensland, Australia, 2010.

[13] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially rep-resentative embedded benchmark suite," in *Proceedings in the 2001 IEEE International Workshop on Workload Characterization*, Austin, TX, USA, 2001, pp. 3–14.