Supporting Cache Coherence in Heterogeneous Multiprocessor Systems

Taeweon Suh, Douglas M. Blough, and Hsien-Hsin S. Lee
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332
{*suhtw, dblough, leehs*}*@ece.gatech.edu*

Abstract

In embedded system-on-a-chip (SoC) applications, the demand for integrating heterogeneous processors onto a single chip is increasing. An important issue in integrating multiple heterogeneous processors on the same chip is to maintain the coherence of their data caches. In this paper, we propose a hardware/software methodology to make caches coherent in heterogeneous multiprocessor platforms with shared memory. Our approach works with any combination of processors that support invalidation-based protocols. As shown in our experiments, up to 58% performance improvement can be achieved with low miss penalty at the expense of adding simple hardware, compared to a pure software solution. Speedup can be improved even further as the miss penalty increases. In addition, our approach provides embedded system programmers a transparent view of shared data, removing the burden of software synchronization.

1. Introduction

Shared memory multiprocessor architectures employ cache coherence protocols such as MSI [2], MESI [16], and Dragon protocol [14], to guarantee data integrity and correctness when data are shared and cached within each processor. For example, IBM's PowerPC755 [10] supports the MEI protocol (**M**odified, **E**xclusive, and **I**nvalid), Intel's IA32 Pentium class [4] processor supports the MESI protocol, to name a few. Several variants of the MESI protocol are used in modern microprocessors, e.g. the MOESI protocol (Exclusive **M**odified, Shared M**O**dified, **E**xclusive Clean, **S**hared Clean, and **I**nvalid) from SUN's UltraSPARC [15] and a slightly different MOESI protocol (**M**odified, **O**wned, **E**xclusive, **S**hared, and **I**nvalid) from the latest AMD64 architecture [1].

Conventionally, commercial servers and high-performance workstations enable multiprocessing capability by integrating homogeneous processors on the platforms. For these systems, it is straightforward to integrate several processors together using a shared bus since the bus interface and the cache coherence protocol are completely compatible. Once homogeneous processors are integrated with a shared bus, the cache coherence is automatically guaranteed through the hardware so long as the cache controller in each processor includes cache coherence functions. However, as system-on-a-chip (SoC) technology becomes prevalent and specific computing capability is demanded in embedded applications, a highly integrated embedded system starts to integrate heterogeneous processors with different instruction set architectures onto a single chip to expedite the processing speed for different algorithms, and hence maximize the throughput. For instance, in real-time embedded systems the MPEG and audio decoding efficiency are essential while the TCP/IP stack processing speed is also critical. Obviously, one general-purpose processor or a single digital signal processor (DSP) alone cannot be sufficient enough in managing the entire system and providing the computational power required. Under such circumstances, one can employ a media processor or a DSP for the MPEG/audio applications while a more general purpose processor for the TCP/IP stack processing which tends to be more control-intensive. To perform these heterogeneous operations seamlessly, the cache coherence issues among these heterogeneous processors should be studied, evaluated, and analyzed.

The design complexity of integrating heterogeneous processors on SoCs is not trivial since it introduces several problems in both design and validation due to different bus interface specifications and incompatible cache coherence protocols. Sometimes it is even worse as some embedded processors do not support cache coherence at all. In this paper, we overcome these issues by proposing a hardware/software methodology and demonstrate physical design examples using three commercially available heterogeneous embedded processors — Write-back Enhanced Intel486 [3], PowerPC755, and ARM920T [12]. Our hardware design was based on Verilog Hardware Description Language. Seamless CVE [9] from Mentor Graphics and VCS [18] from Synopsys were used as the simulation tools.

The rest of this paper is organized as follows. Section 2 overviews prior work. Section 3 discusses our proposed methodology for maintaining cache coherence in a heterogeneous multiprocessor platform. Section 4 presents two implementations based on our methodology. Section 5 shows the performance evaluation, and finally we conclude our work in Section 6.

2. Related Work

Large scale heterogeneous multiprocessing systems contain distributed shared memory. In such a system, a directory-

Table 1: Heterogeneous Platform Classes

Platform (PF)	Cache coherence hardware inside each processor	
	Processor 1 (P1)	Processor 2 (P2)
PF1	No	No
PF2	Yes (No)	No (Yes)
PF3	Yes	Yes

based cache coherence [19] scheme can be used for coherency among distributed shared memory. Two major classes of finding the source of directory information for a block are flat directory schemes and hierarchical directory schemes. Flat directory schemes can be divided into two classes: memory-based schemes and cache-based schemes [6]. These directory-based protocols address the inter-cluster coherence issues of a distributed shared memory system while the bus-based snoop mechanism is used for maintaining intra-cluster coherence. Even though the directory-based protocol can address the coherence issue among homogeneous or heterogeneous clusters, the snoop-based bus protocols, however, fail to address the coherence problem for intra-cluster heterogeneous processors because of the distinct nature of each individual coherence and bus protocol.

In the embedded SoC domain, design methodology for an application specific multiprocessor SoC has been proposed with the concept of a *wrapper* to overcome the problem of incompatible bus protocols [20, 21]. Wrappers allow automatic adaptation of physical interfaces to a communication network. Generic wrapper architectures and automatic generation method have been proposed to facilitate the integration of existing components [13]. Our proposed solution can be incorporated in the wrapper to manage the data coherence among heterogeneous processors.

3. Proposed Approach

There are two main categories of cache coherence protocols: update-based protocols and invalidation-based protocols. In general, invalidation-based strategies are more robust, therefore, most vendors use it as the default protocol [6]. In this paper, we focus our discussion on those processors supporting invalidation-based protocols and study the implication of integrating them with processors without any inherent cache coherence support. Heterogeneous processor platforms can be classified into three classes in terms of the processors' cache coherence support as shown in Table 1, in which we simplify the scenario to a dual-processor platform. Our proposed approach can be easily extended to platforms with more than two processors. For PF1 and PF2, special hardware is needed and there are limitations in the resulting coherence mechanism. These cases are discussed with an example in Section 4. For PF3, the cache coherence can be maintained with simple hardware. We discuss PF3 in this section and Section 4.

Integrating processors with different coherence protocols restricts the use of the entire protocol states. Only the states that are common from distinct protocols are preserved. For example, when integrating two processors with MEI and MESI, the coherence protocol in a system must be MEI. We present methods of integration according to the combination of invalidation-based protocols. We assume

Table 2: Problem and Solution with MEI and MESI

seq.	Read Write on cache line C	No Proposed Solution		With Proposed Solution	
		C state in P1 (MESI)	C state in P2 (MEI)	C state in P1 (MESI)	C state in P2 (MEI)
ⓐ	P1 read	**I ⇒ E**	**I**	**I ⇒ E**	**I**
ⓑ	P2 read	**E ⇒ S**	**I ⇒ E**	**E ⇒ I**	**I ⇒ E**
ⓒ	P2 write	**S(Stale)**	**E ⇒ M**	**I**	**E ⇒ M**
ⓓ	P1 read	**S(Stale)**	**M**	**I ⇒ E**	**M ⇒ I**

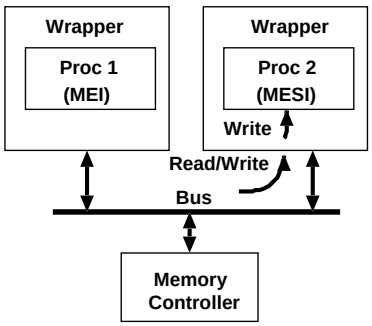

Figure 1: Method to Remove the Shared State

that the cache-to-cache sharing is implemented only in processors supporting the MOESI protocol, as most commercial processors do.

In the subsequent sections, we will discuss protocol integration methods for four major protocols MEI, MSI, MESI, and MOESI. The variations include (1) MEI with MSI/MESI/MOESI, (2) MSI with MESI/MOESI, and (3) MESI with MOESI. Scalability and DMA issues are also discussed in this section.

3.1 MEI with MSI, MESI, or MOESI

Integrating the MEI protocol with others requires the removal of the shared state. To illustrate the problem with the shared state, we use the example in Table 2 assuming that Processor 1 supports the MESI protocol and Processor 2 supports the MEI protocol, with the operation sequence ⓐ ⓑ ⓒ ⓓ executed for the same cache line **C**. Operation ⓐ changes the state from I to E in Processor 1 as a result of the read. ⓑ changes the state from I to E in Processor 2 and from E to S in Processor 1. Since C is in the state E in Processor 2, transaction ⓒ does not appear on the bus even though Processor 1 has the same line in the S state. It invokes the state transition from E to M in Processor 2. However, the state of the cache line in Processor 1 remains the same. Therefore, transaction ⓓ accesses the stale data, which should have been invalidated during ⓒ.

Figure 1 depicts our proposed method to remove the shared state. Since the transition to the shared state occurs when the snoop hardware in the cache controller observes a read transaction on the bus, the way to remove the shared state is simply to convert a "read" operation to a "write" operation within the wrappers of snooped processors. The memory controller should see the actual operation in order to access the memory correctly when it needs to.

Using the MESI protocol as an example, the state change from E to S occurs only when the snoop hardware in the

cache controller sees a read transaction on the bus for the cached line of the E state. Therefore, in order to remove the shared state, it is sufficient for the wrapper to convert every read transaction on the bus to a write during snooping. When the snoop hardware in the cache controller sees a write transaction on a cache line in a modified or an exclusive state, it drains out or invalidates the cache line ("drain" means writing back the modified cache line to memory and invalidating the cache line). In this way, the shared state is excluded in the controllers' state machines.

The last two columns of Table 2 illustrate the state transitions with our proposed solution. The transaction ⓑ invokes the state transaction from E to I in Processor 1 since Processor 1 observes a write operation on the bus. The transaction ⓓ changes the state from M to I in Processor 2 for the same reason. The following subsections detail how the state machines are changed for different invalidation-based protocols with the proposed approach. With the techniques described below, the MSI, MESI, and MOESI protocols are reduced to MEI.

3.1.1 MSI Protocol

In the MSI protocol, two transitions exist to reach the S state: (1) I ⇒ S when the cache controller sees a read miss to a cache line and (2) M ⇒ S when the snoop hardware in the cache controller sees a read operation on the bus. In case (1), the S state cannot be removed since this transition is invoked by its own processor. However, even though it is in the S state, only one processor owns a specific cache line at any point in time because the S state changes to the I state whenever other processors read or write the same cache line. (Note that the wrapper converts a read into a write.) Therefore, despite of the name, the S state is equivalent to the E state. The state transition from M to S cannot occur since the wrapper always converts a read operation to a write operation. Only the M to I transition is allowed with the operation conversion.

3.1.2 MESI Protocol

In the MESI protocol, there are three possible transitions that reach the S state: (1) I ⇒ S when a read miss occurs and the shared signal [19] is asserted, (2) E ⇒ S when the snoop hardware in the cache controller sees a read operation for a clean cache line on the bus, and (3) M ⇒ S when the snoop hardware in the cache controller sees a read operation for a modified (or dirty) cache line on the bus. To remove the S state, the wrapper always deasserts the shared signal. This means transition (1) cannot occur. Transitions (2) and (3) also cannot occur because the wrapper informs the snooped caches of writes for read operations. Therefore, the S state is completely removed.

3.1.3 MOESI Protocol

The same techniques used for the MESI protocol can be applied to the MOESI protocol except the O state needs to be handled. The O state can only be reached when the snoop hardware in the cache controller observes a read operation on the bus for a modified cache line. Nevertheless, the O state is never entered since the cache controller never sees a read operation on the bus when snooping.

3.2 MSI with MESI, or MOESI

In integrating MSI and MESI protocols, the E state must be prohibited. Suppose that Processor 1 supports the MSI protocol and Processor 2 supports the MESI protocol and the operations in Table 3 are executed for the same cache line. ⓐ changes the state from I to S in the Processor 1. ⓑ causes the state transition from I to E in the Processor 2 while the cache line status of Processor 1 remains unchanged because Processor 1 does not assert the shared signal. ⓒ invokes only the E to M transition in Processor 2. As a result, Processor 1 reads the stale data in ⓓ due to a cache hit indicated by the S state. Therefore, the E state should not be allowed in the protocol. Our technique to remove the E state from the MESI protocol is to assert the shared signal whenever a read miss occurs. With this technique, the transaction ⓑ invokes the state transaction from I to S in Processor 2 and the transaction ⓓ changes the state from M to S in Processor 2

Table 3: Problem and Solution with MSI and MESI

seq.	Read Write on cache line C	No Proposed Solution		With Proposed Solution	
		C state in P1 (MSI)	C state in P2 (MESI)	C state in P1 (MSI)	C state in P2 (MESI)
ⓐ	P1 read	**I ⇒ S**	**I**	**I ⇒ S**	**I**
ⓑ	P2 read	**S**	**I ⇒ E**	**S**	**I ⇒ S**
ⓒ	P2 write	**S(Stale)**	**E ⇒ M**	**I**	**S ⇒ M**
ⓓ	P1 read	**S(Stale)**	**M**	**I ⇒ S**	**M ⇒ S**

The same method can be applied to the integration of MSI and MOESI protocols with one additional constraint imposed. In MOESI, the M to O transition occurs when the processor observes a read transaction on the cache line of the M state. Then, a cache-to-cache transition occurs. Since the cache-to-cache sharing is not allowed in the MSI protocol, the M to O transition should not occur. To preclude this transition, the same technique used for eliminating the shared state can be used, i.e. "read"-to-"write" conversion within wrappers. Since the shared signal is always asserted and the read to write conversion should be employed, the E and O state transition never occurs. Therefore, the MOESI protocol is reduced to the MSI protocol.

With these techniques described above, the MESI and MOESI protocols are reduced to MSI.

3.3 MESI with MOESI

To prohibit cache-to-cache sharing while integrating MESI and MOESI protocols, read-to-write conversion can again be employed. This precludes the transitions from E to S and from M to O in MOESI protocol. However, the I to S transition is allowed. Therefore, the MOESI protocol is reduced to MESI even though not all of the transitions in MESI are allowed.

3.4 Scalability

Scalability is a function of available bus bandwidth, delay, and implementation cost. For a shared bus system, so long as the bus bandwidth is constant, adding additional processors to the system, regardless of homogeneous or heterogeneous, the scalability will be always constrained by the available bus bandwidth. With respect to the extra delays incurred in our proposed approach against its homogeneous counterparts, they are associated with the wrapper and the arbiter and can be implemented rather inexpensively. In the wrappers, only read-to-write

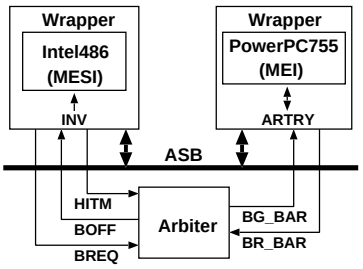

Figure 2: PowerPC755, Intel486 coherence

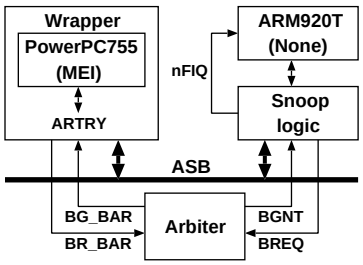

Figure 3: PowerPC755, ARM920T coherence

conversion and/or shared signal assertion/de-assertion are needed. The arbiter is designed to coordinate and manage the bus ownership for the requests upon a snoop-hit from these wrappers. It uses a prority-based decision logic to grant the bus for the snoop-hit processor. Our preliminary synthesized design using 0.35μm technology shows a merely 0.74ns increase of the delay for an arbiter with 4 heterogeneous processors. This timing impact is fairly marginal, thus will not impede the scalability.

3.5 DMA

In general, memory-mapped I/Os are allocated in uncacheable memory space. Therefore, DMA should not cause any coherence issue. For some unconventional systems that allow DMA to transfer data between cacheable regions, however, the coherence problem can be resolved by allowing DMA controller to concede the bus mastership whenever a snoop hit occurs during DMA operations and reclaim it after writeback if the corresponding cache line is dirty.

4. Case Study

In this section, we present two implementations using commercially available embedded processors: PowerPC755, Write-back Enhanced Intel486 (hereafter, Intel486 is used for Write-back Enhanced Intel486), and ARM920T. PowerPC755 uses the MEI protocol, Intel486 supports a modified MESI protocol, and no cache coherence is supported in ARM920T. A multiprocessor platform employs a shared bus for data transactions between main memory and processors. Several bus architectures for SoC were proposed by industry, for example, IBM's CoreConnect bus architecture [5], Palmchip's CoreFrame [8], and ARM's Advanced Microcontroller Bus Architecture (AMBA) [11]. A common characteristic among these architectures is that they use two separate pipelined buses: one for high speed devices and one for low speed devices. In this paper, we study the Advanced System Bus (ASB), an AMBA bus, as the shared bus protocol. The AMBA is one of the most popular bus protocols in embedded system design [7].

As shown in Figure 2, the schematic diagram integrates a PowerPC755 and an Intel486, representing a case of the PF3. Wrappers are needed for the protocol conversions between the processors' buses and the ASB, in addition to read operation conversion. On the PowerPC755 side, the conversion from a read operation to a write operation is not needed since the S state is not present in the state machine, whereas the S state should be removed on the Intel486 side by asserting the INV input signal, a cache coherency protocol pin. It is sampled on snoop cycles by the Intel486 cache controller. If it is asserted, the cache controller invalidates an addressed cache line if the cache line is in the E or S state. If it is in the M state, the line is drained out to memory. Normally, INV is de-asserted on read snoop cycles and asserted on write snoop cycles. To remove the S state, it should be asserted on both read and write snoop cycles. In the Intel486's cache, cache lines are defined as write-back or write-through at allocation time. Only write-through lines can have the S state, and only write-back lines can have the E state. Therefore, the protocol for write-through lines is the SI protocol while the protocol for write-back lines is the MEI protocol. When a snoop hit occurs on the M state line of the Intel486 cache, the HITM output signal is asserted and the wrapper around the PowerPC755 informs the core of a snoop hit by asserting the ARTRY (Address Retry) input signal. Then, the PowerPC755 immediately yields the bus mastership to the Intel486 so the cache controller in the Intel486 drains out the modified line to memory. When a snoop hit occurs on the M state line of the PowerPC755 data cache, the PowerPC755 asserts the ARTRY output signal and the arbiter immediately asserts BOFF so the Intel486 yields the bus mastership to the PowerPC755. Then, the cache controller in the PowerPC755 drains out the modified line to memory.

Figure 3 shows another example of a heterogeneous platform using PowerPC755 and ARM920T representing a case of PF2. The same methodology used in ARM920T is applicable to PF1. The wrapper in the figure converts the PowerPC bus protocol to the ASB protocol, and vice versa. It also allows the PowerPC755 to monitor the bus transactions generated by the ARM920T. The snoop logic provides snooping capability for the ARM920T, which does not have any native cache coherence support. It keeps all the address tags of the ARM920T's data cache inside a content addressable memory (or TAG CAM) watching bus transactions initiated by the ARM920T. When the tag of a requested address generated by the PowerPC755 matches an entry of the TAG CAM, it triggers a snoop hit to the ARM920T by asserting a fast interrupt (nFIQ). An interrupt service routine is responsible for draining the snoop-hit cache line if the line is modified or invalidating it if the line is clean.

Even though this architecture can make caches coherent, the delay in responding to interrupts can have an effect on how the coherence protocol executes. Depending on the processors and bus protocols used, this could result in some limitations of the approach. For example, we had to impose restrictions on the way locks were implemented by the system and used by the programmer in order for

Table 4: Simulation Environment

Simulators		• Seamless CVE • VCS
Operating frequencies		• PowerPC755: 100MHz • ARM920T: 50MHz • ASB: 50MHz
Instruction caches		Enabled
Data caches		• Private data: Enabled • Shared data: Selectively enabled
Memory access time	**Single Word**	6 cycles
	Burst (8 words)	• 6 cycles for 1^{st} word • 1 cycle for each subsequent word

this platform to work properly.

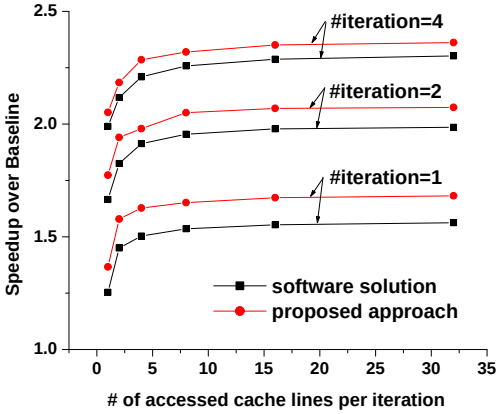

Figure 4: Worst Case Results

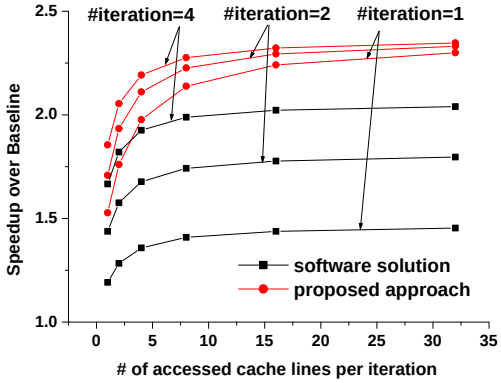

Figure 5: Best Case Results

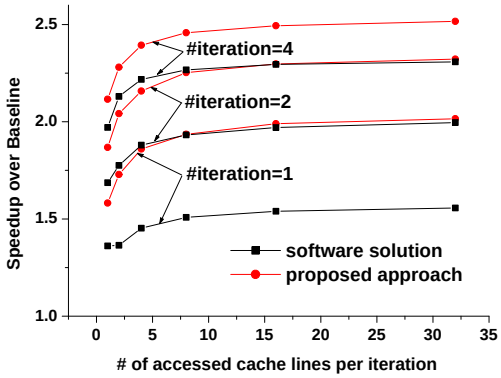

Figure 6: Typical Case Results

5. Performance Evaluation

Simulations were performed using a worst-case scenario (WCS), a typical-case scenario (TCS), and a best-case scenario (BCS) microbench programs. In the microbench programs, one task runs on each processor. Each task will try to access a critical section (i.e. shared memory), which is protected by a lock mechanism. Once a task acquires the lock, it accesses a number of cache lines and modifies them for different number of iterations before exiting the critical section. The microbench program was implemented with each task acquiring the lock alternately, which means the simulation assumes the worst-case situation for lock acquisition and releasing.

First, we use a machine that disables the data caches for the shared area as our *baseline* system. Also note that the coherence can be achieved via software synchronization, a complete *software solution*, for a shared-memory system with caches. As such, the programmers are responsible for draining or invalidating all the used cache lines in the critical section before exiting the critical section. The simulation environment and the hardware configurations are summarized in Table 4. The platform with the PowerPC755 and ARM920T is used to quantify the performance. The Intel486 and PowerPC755 platform[1] should outperform the PowerPC755 and ARM920T platform due to the absence of an interrupt service routine.

Simulations of each alternate solution were performed for each scenario to evaluate the performance of our approach. Lock variables are not cached in all simulations. Figure 4 to 6 show the speedup of the software solution and our proposed hardware approach normalized to the baseline for different numbers of iterations. Figure 4 shows the WCS results. In the WCS, two tasks keep accessing the same blocks of memory. Our proposed solution shows 136% performance improvement compared to the baseline when #iterations=4. It also shows better performance than the software solution by at least 2.56% for all WCS simulations.

In the BCS, the ARM920T accesses the critical section whereas the PowerPC755 does not. The ARM920T drains out the used blocks before exiting the critical section in the software solution, but it does not need to drain out the used blocks in the proposed solution. The results in Figure 5 show that the speedup increases as the number of accessed cache line increases. Simulation with 32 cache lines shows 58.2% improvement over the software solution with #iterations = 1.

In the TCS, each task randomly picks up shared blocks of memory among 10 blocks before getting into the critical section. Figure 6 shows the simulation results. Simulation with 32 cache lines shows 29.5% speedup compared to the software solution with #iterations = 1.

[1]Due to some unavailable capability in our simulation tools, the results of a PowerPC755/Intel486 system are not reported. This will be in our future work.

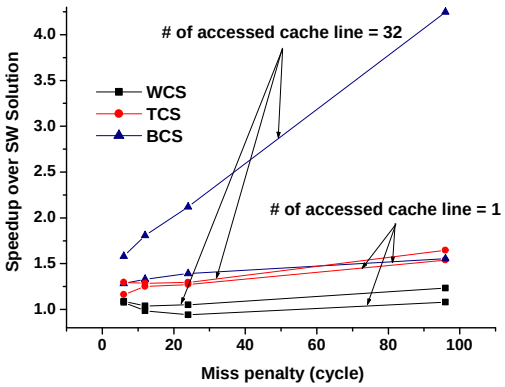

Figure 7: Results according to Miss Penalty

So far, we have assumed that memory access time is fixed to 6 cycles for a single word access, and 13 cycles for a burst access as shown in Table 4. Figure 7 shows the performance results as the miss penalty (memory access time) increases. Note that the software solution is used as the baseline in this figure. As the miss penalty increases, the performance difference also increases in favor of our approach with a few exceptions in the WCS. These exceptions come from cache line replacements and/or interrupt processing overheads that vary as the miss penalty changes. These exceptions are expected to be removed in PF3 since the interrupt service routine is not needed. The BCS result with 32 cache lines shows a 4.24x speedup compared to the software solution when the miss penalty is increased to 96 cycles.

6. Conclusions

In this paper, we presented a methodology to maintain the coherence of data caches in heterogeneous processor platforms. Cache coherence can be guaranteed simply by implementing wrappers in platforms where processors support any invalidation protocol. Read to write operation conversion and/or shared signal are used within wrappers to maintain coherence depending on combination of coherence protocols. The integrated coherence protocol will at most consist of all the common states from various protocols in a system. Using commercial embedded processors as the experimental platforms, our simulation results showed 58% performance improvement for low miss penalties and 324% performance improvement for higher penalties at the expense of simple hardware, compared to a pure software solution. Platforms without need for a special interrupt service routine would perform even better. As the miss penalty increases, the speedup also increases in favor of our approach. As heterogeneous processor SoCs become more prevalent in future system design, our methodology will be very useful and effective for integrating heterogeneous coherence protocols in the same system. In the future, we plan to apply our approach to emerging technologies that tightly integrate between a main processor and specialized I/O processors such as network processors [17].

7. References

[1] AMD. AMD64 Technology. http://www.amd.com/usen/assets/content_type/white_papers_and_tech_docs/24593.pdf.

[2] F. Baslett, T. Jermoluk, and D. Solomon. The 4D-MP Graphics Superworkstataion: Computing+Graphics= 40MIPS+40MFLOPS and 100,000 Lighted Polygons per Second. In *Proceedings of the COMPCON'88*, pages 468–471, 1988.

[3] Intel Corp. Embedded Intel486 Hardware Reference Manual. http://www.intel.com/design/intarch /manuals/273025.htm.

[4] Intel Corp. The IA32 Intel Architecture Software Developer's Manual. http://developer.intel.com/design/ pentium4/manuals/245472.htm.

[5] IBM Corporation. CoreConnect Bus Architecture. http://www.chips.ibm.com/ products/coreconnect.

[6] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, 1999.

[7] Embedded.com. http://www.embedded.com/ story/OEG20021204S0005.

[8] B. Gordan. An Efficient Bus Architecture for System-on-a-Chip Design. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, pages 623–626, May 1999.

[9] Mentor Graphics. Hardware/Software Co-Verification: Seamless. http://www.mentor.com/seamless.

[10] Motorola Inc. MPC 750A RISC Microprocessor Hardware Specification. http://www.mot.com/SPS/PowerPC /library/750_hs.pdf.

[11] ARM Ltd. AMBA Specification Overview. http://www.arm.com/ Pro+Peripherals/AMBA.

[12] ARM Ltd. ARM920T Technical Reference Manual. http://www.arm.com/arm/ documentation?OpenDocument.

[13] D. Lyonnard, A. Baghdadi S. Yoo, and A. A. Jerraya. Automatic Generation of Application-Specific Architectures for Heterogeneous Multiprocessor System-on-Chip. In *Proceedings of the 38th Conference on Design Automation*, 2001.

[14] E. McCreight. The Dragon Computer System: An Early Overview. Technical report, Xerox Corp., 1984.

[15] Sun Microsystems. UltraSPARC User's Manual. http://www.sun.com/processors/manuals /802-7220-02.pdf.

[16] M. Papamarcos and J. Patel. A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 348–354, 1984.

[17] D.-S. Sun and D. M. Blough. I/O Threads: A Novel I/O Approach for System-on-a-Chip Networking. Technical report, CERCS, Georgia Institute of Technology, 2003.

[18] Synopsys. VCS Data Sheet. http://www.synopsys.com/products/simulation /vcs_ds.html.

[19] C. K. Tang. Cache Design in the Tightly Coupled Multiprocessor System. In *AFIPS Conference Proceedings of National Computer Conference*, pages 749–753, 1976.

[20] S. Vercauteren, B. Lin, and H. De Man. Constructing Application-Specific Heterogeneous Embedded Architectures from Custom HW/SW Applications. In *Proceedings of the 33rd Conference on Design Automation*, June 1996.

[21] S. Yoo, G. Nicolescu, D. Lyonnard, A. Baghdadi, and A. A. Jerraya. A Generic Wrapper Architecture for Multi-Processor SoC Cosimulation and Design. In *CODES/CASHE*, 2001.