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, Georgia 30332-0250
{suhtw, doug.blough, leehs}@ece.gatech.edu

Abstract

In embedded system-on-a-chip (SoC) applications, the need for integrating heterogeneous processors in a single chip is increasing. An important issue in integrating heterogeneous processors is how to maintain the coherence of 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 any invalidation-based protocol. As shown in our simulations, up to 38% speedup can be achieved with a 13-cycle miss penalty at the expense of 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 [1], MESI [2], and Dragon protocol [3], to guarantee data integrity and correctness when data are shared and cached within each processor. For example, IBM's PowerPC755 [4] supports the MEI protocol (**M**odified, **E**xclusive, and **I**nvalid), Intel's IA32 Pentium class [5] 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 [6] and a slightly different MOESI protocol (**M**odified, **O**wned, **E**xclusive, **S**hared, and **I**nvalid) from the most recent AMD64 architecture [7].

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 more computing power is demanded in embedded applications, highly integrated embedded systems would integrate a few heterogeneous processors, with different instruction set architectures, on a single chip to expedite the processing speed and 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 will be ineffective 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 different one for the TCP/IP stack processing. 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 non-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 inherently. 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 [8], PowerPC755, and ARM920T [9]. Our hardware design was based on Verilog Hardware Description Language. Seamless CVE [10] from Mentor Graphics® and VCS [11] from Synopsys® were used as the simulation tools.

This paper is organized as follows. Section 2 discusses our proposed methodology for maintaining cache coherence in heterogeneous multiprocessor systems. Section 3 presents two viable implementations. Section 4 shows the performance evaluation, and finally we conclude our work in Section 5

2. Proposed approach

There are two main categories of cache coherence protocols: update-based protocols and invalidation-based protocols. In general, invalidation-based strategies have been found to be more robust and are therefore provided as the default protocol by most vendors [12]. In this paper, we focus our discussion on those processors that support 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 platforms 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. In 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 3. In PF3, the cache coherence can be maintained with simple hardware. We discuss PF3 in this section and Section 3. Integrating processors with different coherence protocols restricts the usage of the entire protocol states. Only the states that are common from distinct protocols can be 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 that the cache-to-cache sharing is implemented only in processors supporting the MOESI protocol, as most commercial processors do.

Table 1: Heterogeneous platform classes

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

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.

2.1 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.

Table 2: Problem with MEI and MESI

seq	Operation on cache line C	C state in Processor 1 (MESI)	C state in Processor 2 (MEI)
ⓐ	Processor 1 reads	**I→E**	**I**
ⓑ	Processor 2 reads	**E→S**	**I→E**
ⓒ	Processor 2 writes	**S (stale)**	**E→M**
ⓓ	Processor 1 reads	**S (stale)**	**M**

ⓐ changes the state from I to E in Processor 1. ⓑ changes the state from I to E in Processor 2 and from E to S in Processor 1. Since the cache line is in the E state 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.

Figure 1 illustrates 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 wrappers of snooping processors. The memory controller should see the actual operation in order to access the memory correctly when it needs to do.

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 a 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 following subsections describe how the state machines are changed for different invalidation-based protocols with the proposed approach.

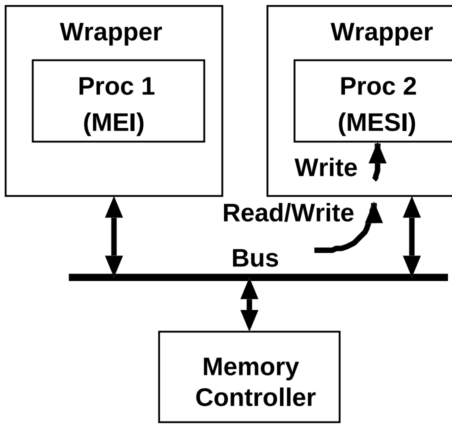

Figure 1: The method to remove the shared state

2.1.1 MSI protocol. In the MSI protocol, two transitions exist to reach the S state: (1) I $\rightarrow$ S when the cache controller sees a read miss to a cache line and (2) M $\rightarrow$ 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 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.

2.1.2 MESI protocol. In the MESI protocol, there are three possible transitions that reach the S state: (1) I $\rightarrow$ S when a read miss occurs and the shared signal [12] is asserted, (2) E $\rightarrow$ S when the snoop hardware in the cache controller sees a read operation for a clean cache line on the bus, and (3) M $\rightarrow$ 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 de-asserts the shared signal. This means transition (1) cannot occur. Transitions (2) and (3) also cannot occur because the wrapper informs snooping caches of writes for read operations. Therefore, the S state is completely removed.

2.1.3 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.

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

2.2 MSI with MESI, or MOESI

In integrating MSI and MESI protocols, the E state is not allowed. 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.

Table 3: Problem with MSI and MESI

seq	Operation On cache line C	C state in Processor 1 (MSI)	C state in Processor 2 (MESI)
ⓐ	Processor 1 reads	**I→S**	**I**
ⓑ	Processor 2 reads	**S**	**I→E**
ⓒ	Processor 2 writes	**S (stale)**	**E→M**
ⓓ	Processor 1 reads	**S (stale)**	**M**

ⓐ changes the state from I to S in the Processor 1. ⓑ makes the state transition from I to E in the Processor 2 while the cache line status of Processor 1 remains unchanged because Processor 1 cannot 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.

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.

2.3 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. 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 [13], Palmchip's CoreFrame [14], and ARM's Advanced Microcontroller Bus Architecture (AMBA) [15]. 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 [16].

Figure 2 shows the schematic diagram for the integration of PowerPC755 and Intel486, which represents a case of the PF3.

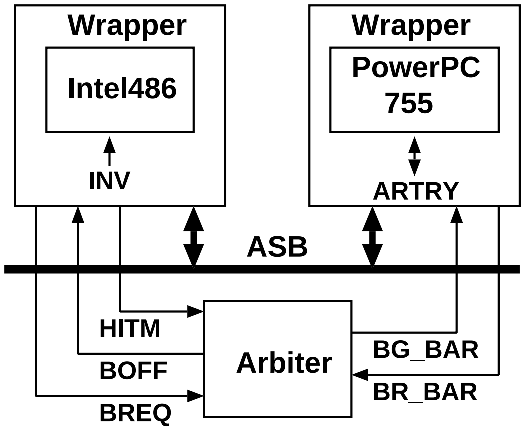

Figure 2: PowerPC755, Intel486 coherence

Wrappers are needed for the protocol conversions between the processors' buses and 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 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 can be employed in PF1.

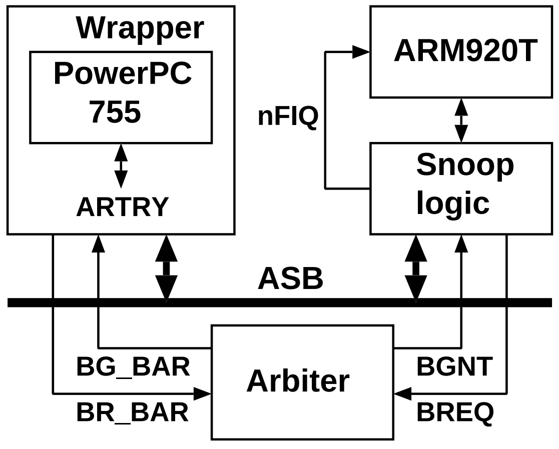

Figure 3: PowerPC755, ARM920T coherence

The wrapper 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, there is one limitation when at least one of the processors in a heterogeneous processor platform does not have cache coherence hardware such as in the case of ARM920T. Therefore, PF1 and PF2 have this limitation. Figure 4 illustrates the problem. Suppose that lock variables and the shared data are allowed to be cached, and the shared data 1 is currently in the data cache of the ARM920T. After acquiring the lock, PowerPC tries to access the shared data as shown in (1). Therefore, a snoop hit occurs in the snoop logic, and the nFIQ is asserted as illustrated in (2). Then, ARM is supposed to drain out/invalidate the cache line in the interrupt service routine. However, ARM may or may not respond to the interrupt immediately, depending on the status of the CPU pipeline. During the interrupt response time as shown in (3), ARM may try to check the lock to see if it is released. Lock variables are currently in the data cache of PowerPC, since PowerPC has the lock. Therefore, PowerPC should drain out the cache line, where lock variables are stored. However, if PowerPC gets the bus mastership, it is supposed to retry the transaction, which it did in (1), instead of draining out the lock variables. We call this situation "the hardware deadlock".

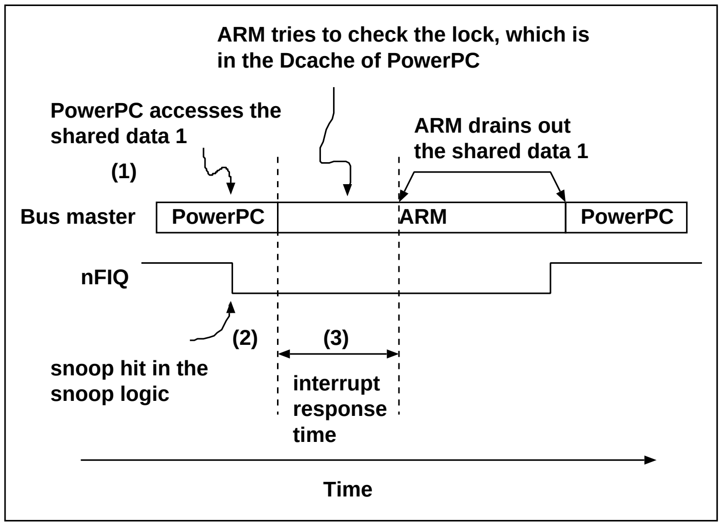
ARM tries to check the lock, which is in the Dcache of PowerPC
PowerPC accesses the shared data 1
(1)
ARM drains out the shared data 1
Bus master
PowerPC
ARM
PowerPC
nFIQ
(2)
(3)
snoop hit in the snoop logic
interrupt response time
Time

Figure 4: Hardware deadlock problem

There are two solutions to avoid the hardware deadlock problem.

- Do not cache the lock variables
- Hardware lock register in a system [17]

In the first solution, a software lock using, for example, the Bakery algorithm [18] can be implemented. In this case, the programmer is restricted to perform all shared variable operations within critical sections, or a similar deadlock can occur on non-lock variables. In the second solution, a simple but special hardware needs to be designed. This hardware has only 1-bit lock register, and sits on a shared bus. Since the lock variables are not cached, the hardware deadlock does not occur in either case. For the same reason, the system can have only one lock.

4. Performance evaluation

Simulations were performed with the worst-case scenario (WCS), the typical-case scenario (TCS), and the best-case scenario (BCS) microbench programs. In the microbench programs, one task runs on each processor. Each task tries to access a critical section (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 exec_time iterations before exiting the critical section. The microbench program was implemented with each task acquiring the lock alternatively, which means the simulation assumes the worst-case situation for lock acquisition/releasing.

Table 4: Simulation environments

Simulators		• Seamless CVE • VCS
Operating frequencies		• PowerPC755: 100MHz[1] • ARM920T: 50MHz[1] • ASB: 50MHz
Instruction caches		Enabled
Data caches		• Private data: Enabled • Shared data: Selectively enabled[2]
Memory access time	**Single word**	6 cycles
	Burst (8 words)	• 6 cycles for the 1st word • 1 cycle for each subsequent word

[1] These low operating frequencies are due to the limitation of simulation models. Similar results are expected for simulations with higher operating frequencies.
[2] Enabled in the proposed solution and software solution

If the hardware does not support cache coherence, two alternate solutions can be used to make data caches coherent:

- Data caches can be disabled for shared data
- Software synchronization can be used with data caches enabled for shared data (software solution)

In the software solution, the programmer should make sure to drain/invalidate 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 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 as a baseline to evaluate the performance of our approach. Lock variables are not cached in all simulations. Figure 5~7 show the execution time with respect to the one with the data cache disabled. Figure 5 shows the WCS results. In the WCS, two tasks keep accessing the same blocks of memory. The proposed solution shows 57.66% performance improvement against the case with data cache disabled when exec_time=4. It also shows better performance than the software solution by at least 2.51% for all WCS simulations.

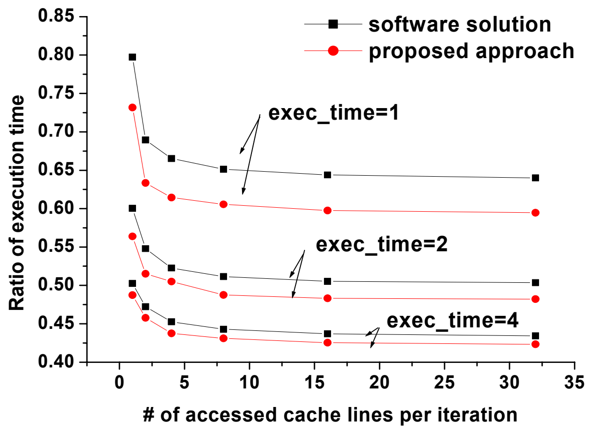

Figure 5: Worst case results

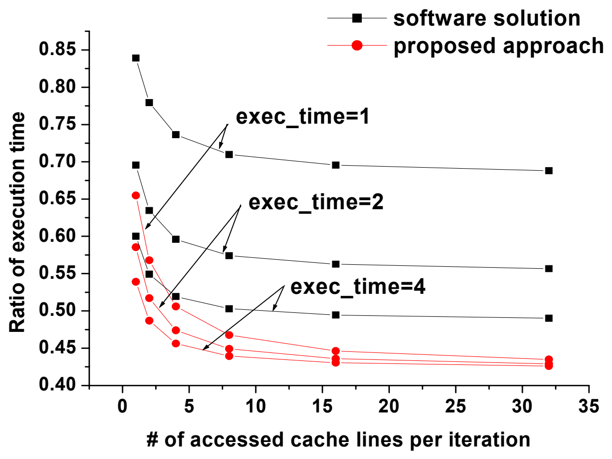

Figure 6: Best case results

In the BCS, the ARM920T accesses the critical section, but PowerPC755 does not access it. 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 6 show that speedup increases as the number of accessed cache line increases. Simulation with 32 cache lines shows 38.22% speedup compared to the software solution with exec_time = 1.

In the TCS, each task randomly picks up shared blocks of memory among 10 blocks before getting into the critical section. Figure 7 shows the simulation results. Simulation with 32 cache lines shows 22.88 % speedup compared to the software solution with exec_time = 1

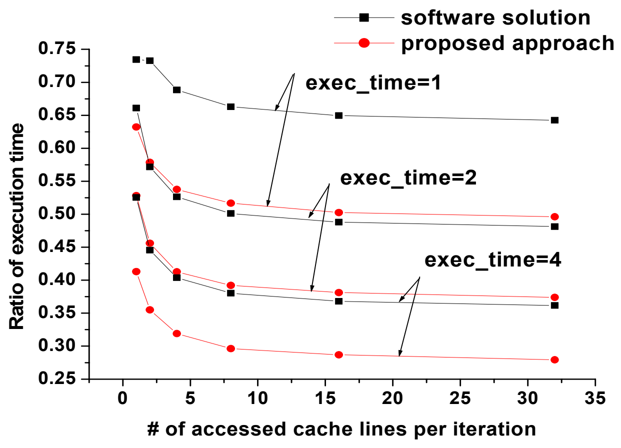

Figure 7: Typical case results

So far, we have assumed that memory access time is fixed to 6 cycles for a single word access, and 13 cycles for burst access as shown in Table 4. Figure 8 shows the simulation results as the miss penalty (memory access time) increases.

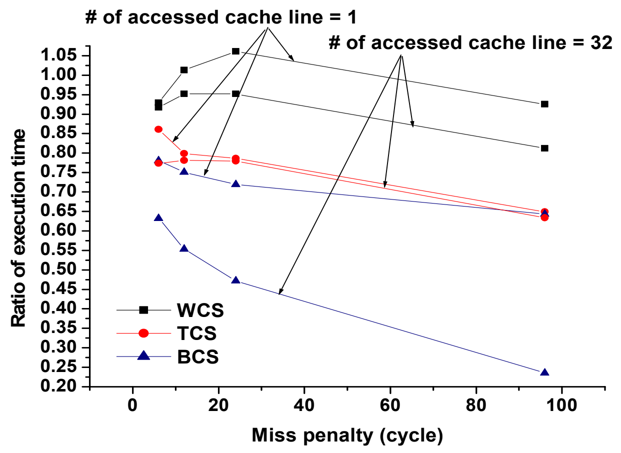

Figure 8: Results according to miss penalty

It shows the relative execution time with respect to the software solution. As the miss penalty increases, the performance difference also increases in favor of our approach with a few exceptions in the WCS and TCS. 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 Simulation with 32 cache lines shows 76.56 % speedup compared to the software solution when the miss penalty is increased to 96 cycles.

5. Conclusion

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 38% speedup for low miss penalties and 76% speedup 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 [19].

6. References

[1] F. Baslett, T. Jermoluk, and D. Solomon, "The 4D-MP Graphics Superworkstataion: Computing+Graphics= 40MIPS+40MFLOPS and 100,000 Lighted Polygons per Second," Proc. 33rd IEEE Computer Society Int'l Conference – COMPCON`88, pp 468-471, February 1988.

[2] M. Papamarcos and J. Patel, "A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories," Proc. 11th Annual Int'l Symposium on Computer Architecture, pp 348-354, June 1984.

[3] E. McCreight, "The Dragon Computer System: An Early Overview," Tech. report, Xerox Corp, September 1984.

[4] Motorola Inc., MPC 750A RISC Microprocessor Hardware Specification, [Online document], Available HTTP: http://www.mot.com/SPS/PowerPC/library/ 750_hs.pdf

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

[6] Sun Microsystems, "UltraSPARC™ User's manual," [Online document], Available HTTP: http://www.sun.com/ processors/manuals/802-7220-02.pdf

[7] AMD, “AMD64 Technology,” [Online document], Available HTTP: http://www.amd.com/usen/assets/content _type/white_papers_and_tech_docs/24593.pdf
[8] Intel Corp., Embedded Intel486TM Hardware Reference Manual, [Online document], Available HTTP: http://www.intel.com/design/intarch/manuals/273025.htm
[9] ARM Ltd., AM920T Technical Reference Manual, [Online document], Available HTTP: http://www.arm.com/arm/ documentation?OpenDocument
[10] Mentor Graphics, Hardware/Software Co-Verification: Seamless, [Online document], Available HTTP: http://www.mentor.com/seamless
[11] Synopsys, “VCS data sheet,” [Online document], Available HTTP:http://www.synopsys.com/products/simulation/vcs_ds.html
[12] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta, Parallel Computer Architecture: A hardware/software approach, Morgan Kaufmann Publishers, Inc., 1999, ch 5.
[13] IBM Corporation. CoreConnect Bus Architecture, [Online document], Available HTTP: http://www.chips.ibm.com/ products/coreconnect
[14] B. Gordan, “An Efficient Bus Architecture for System-on-a-Chip Design,” Proceedings of IEEE Custom Integrated Circuits Conference, pp.623-626, May 1999.
[15] ARM Ltd., “AMBA Specification Overview,” [Online document], Available HTTP: http://www.arm.com/ Pro+Peripherals/AMBA
[16] Embedded.com, [Online article], Available HTTP: http://www.embedded.com/story/ OEG20021204S0005
[17] B. E. S. Akgul and V. J. Mooney, "The System-on-a-Chip Lock Cache," International Journal of Design Automation for Embedded Systems, 7(1-2), September 2002, pp. 139-174
[18] A. Silberschatz, P. Galvin, and G Gagne, Operating System Concepts, 2002, pp 196
[19] Di-shi Sun and Douglas M. Blough, CERCS Technical Report, “I/O Threads: A Novel I/O Approach for System-on-a-Chip Networking,” Georgia Tech, Sep. 2003.