Cache Coherence Support for Non-Shared Bus Architecture on Heterogeneous MPSoCs

Taeweon Suh
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332
suhtw@ece.gatech.edu

Daehyun Kim
Microprocessor Technology Labs
Intel Corporation
Santa Clara, CA 95052
daehyun.kim@intel.com

Hsien-Hsin S. Lee
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332
leehs@ece.gatech.edu

ABSTRACT

We propose two novel integration techniques — *bypass* and *bookkeeping* — in the memory controller to address the cache coherence compatibility issue of a non-shared bus heterogeneous MPSoC. The bypass approach is an inexpensive and efficient solution for computation-bound applications while the bookkeeping approach eliminating unnecessary forwarding traffic offers an alternative for bandwidth-limited applications. Our RTOS kernel simulations show up to 6.65x speedup over the conventional software solution.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-purpose and application-specific systems, Real-time and embedded systems

General Terms

Design

Keywords

Cache coherence, Inter-processor communication, Heterogeneous MPSoC, Real-time and embedded systems

1. INTRODUCTION

Today's SoC designs increasingly demand the flexibility for coping with evolving applications, which leads to the integration of multiple (homogeneous or heterogeneous) embedded processors on MPSoCs. As more processors are integrated in a system, software development will become more expensive and time-consuming. Integrating heterogeneous processors based on a shared bus creates compatibility issues due to the protocol incompatibility of multiple bus interfaces. Sometimes the wrapper [14] can be used to alleviate the problem partially and other approaches such as *Open Core Protocol (OCP)* propose highly configurable interfaces for tackling the same problem. To fully support the functionality of heterogeneous processors, a shared bus protocol should provide a superset of the interface protocols from all processors. In embedded system designs, however, hardware designers are typically reluctant to develop their own superset bus protocol from scratch. Instead, off-the-shelf bus protocols such as AMBA, CoreConnect, and OCP are used but at the cost of compromising the versatility provided by each processor. As a result, very few MPSoC vendors use a shared bus approach. Commercial MPSoCs such as TI's OMAP or Philip's Nexperia [13] employ multiple bus architectures to fully utilize the protocols based on applications' demands. As such, an explicit software-based synchronization mechanism must be used, leading to performance degradation caused by inter-processor communication.

Another critical demand in SoC designs is to meet real-time constraints of embedded applications. Real-time operating systems (RTOS) are commonly used in embedded systems. A hard real-time system provides a guarantee that the response to an event must happen within a specified deadline while a soft real-time system attempts to do its best to meet the timing constraint. Regardless of any real-time system, given a hardware system, software engineers make an effort to optimize software to meet real-time requirements. As described, the flexibility requirement puts more pressure on the software-side for real-time demands. During a design cycle, if a software optimization fails, it would cause a redesign of the hardware system architecture for meeting real-time constraints. Hardware-software co-design often detects design issues in the early stage to eliminate as many potential problems as possible. Nonetheless, if a software optimization fails, the burden eventually falls on to the hardware side, forcing hardware engineers to trade off the flexibility by employing hardcore IPs for time-critical circuit blocks.

In this paper, we address the inter-processor communication issues on heterogeneous multiprocessors with a multiple bus SoC architecture by proposing low-cost techniques which enable cache coherence capability. Our architectural support enhances system performance easing the tight timing budget for real-time system developments. The rest of this paper is organized as follows. Section 2 discusses prior art. Section 3 details our proposed techniques. We then evaluate our techniques in Section 4 and analyze our simulated results in Section 5. Section 6 concludes this work.

2. RELATED WORK

Techniques for integrating cache coherence protocols on a shared bus-based heterogeneous MPSoC have been proposed in [9, 10, 11] in which *read-to-write conversion* and

DAC 2005, June 13–17, 2005, Anaheim, California, USA.

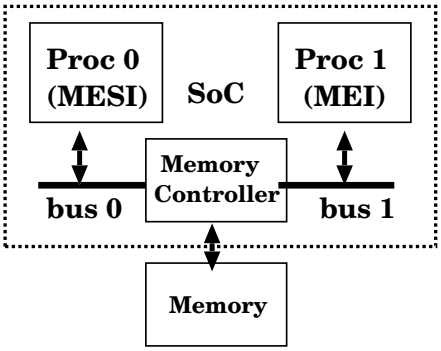

Figure 1: MPMB SoC architecture

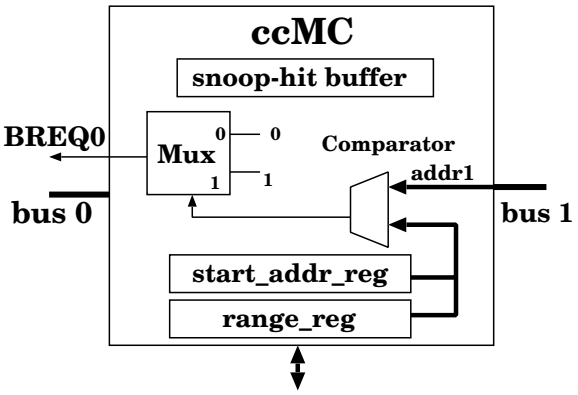

Figure 2: The bypass approach

shared signal assertion/deassertion were implemented in the wrappers around the processors. For maintaining compatibility, the integrated protocol utilizes the common states of the heterogeneous coherence protocols in a system. To further reduce the performance loss due to the lost states, *region-based cache coherence* was introduced in [10, 11] to enable the use of the homogeneous coherence protocols in the shared memory regions.

In distributed shared memory (DSM) systems, directory-based cache coherence protocols are employed in several designs from academia [2, 7, 5] and industry [6, 4, 1]. They eliminate bus snooping that diminishes scalability in a large-scale shared memory system. In a DSM system, the memory controller maintains all cache state information in a common directory to perform appropriate cache coherence actions without a snooping logic. Our bookkeeping scheme proposed in this paper is conceptually similar to this approach. Despite of this, our purpose is not to eliminate snooping for scalability, but rather to filter out unnecessary communication for achieving better bus utilization. In addition, we propose a *region-based directory* scheme that incurs less hardware overhead while exploiting the benefit of a directory-based scheme.

3. INTEGRATION TECHNIQUES

The overall cost of fabricating SoCs or ASICs highly depends on the die budget and the available pin count in packaging technology. In most of the MPSoC systems, the majority of the pins are dedicated to memory interfaces. Given several address and data buses of multiple processors on an MPSoC, dozens of pins are easily consumed for each memory interface. For this reason, architects often make an effort to share or multiplex the memory interface among processors in SoC designs as long as the performance meets its requirement. For example, the C55x DSP and the ARM925T core in TI's OMAP 5910 processor share a common external DRAM interface in a way similar to the schematic shown in Figure 1. In this paper, we target a similar multiprocessor and multiple bus (MPMB) SoC architecture, on which a common memory interface is shared among multiple bus agents (e.g. processors) while each processor uses its own private bus to access the shared memory. To simplify our subsequent discussion, we assume only one processor is connected to each bus as depicted in Figure 1. Note that it can be easily extended to the scenario with multiple buses and multiple processors on each bus where the cache coherence on each bus can be guaranteed by employing the techniques described in [10].

Unlike a shared bus architecture, the processors in a non-shared bus MPSoC system cannot use their native snooping mechanisms because the buses are separated. Without any hardware support, the communication between processors could become very inefficient since the software has to flush out all shared data cache lines every time a processor exits a critical section. For accelerating data communication in MPSoCs, we propose a *cache coherence-enforced memory controller* (ccMC), which wakes up the native snooping capability of each processor. As shown in Figure 2 and Figure 3, a memory-mapped register pair is required inside ccMC. The start_addr_reg is set to the starting address of a shared memory and the range_reg specifies the size of a shared memory region. According to applications' needs, the register pair can be replicated to accommodate more discrete shared regions. We studied two different approaches depending on the allowable silicon budget. The first approach is to bypass a memory request (e.g., from bus1 in Figure 2) blindly to the bus on the other side (bus0) if the requested address falls into the shared memory range specified in the register pair. We call this the *bypass approach*. The second approach keeps track of coherence states of the shared memory blocks inside ccMC. Depending on the state information, ccMC either bypasses a transaction to the other bus or sends a request directly to the main memory, which we call the *bookkeeping approach*. The bookkeeping approach is similar to the DSM's directory-based scheme in a sense that it keeps the directory information in the memory controller. However, it is different from the directory-based scheme since the goal is not to eliminate snooping, but to help snooping for improving bus utilization. Furthermore, ccMC employs a small table to cover shared memory regions based on the applications' needs.

3.1 Bypass approach

The bypass approach bypasses a shared memory request indiscriminately to the other side bus for snooping. For example, suppose that Processor 1 (P1) from the right-hand side of Figure 2 requests a "shared" memory block and misses P1's data cache. Subsequently, this transaction is put on bus1 and ccMC compares the address against the register pair. Since it is in the shared memory range, ccMC then bypasses it to bus0 after being granted the ownership of bus0, in response to the bus request (BREQ0) generated by the comparison match. Finally, Processor 0 (P0) will be able to snoop the bypassed transaction.

The bypass approach consumes bus bandwidth on both sides if a requested address is in the shared memory range. This overhead is due to that ccMC must claim the bus mastership of the other side bus whenever a processor requests a shared data regardless the other processor has the data in its cache or not. The advantage of using the bypass approach is its simplicity and its minimum hardware required. As illustrated in Figure 2, only two comparators, two multiplexers and one register pair are needed.[1] The bypass approach

[1]We only show the schematic diagram from bus1 to bus0 in

Table 1: Problem of the Exclusive state and Solution in the bookkeeping approach

seq.	Operation on cache line C	Without shared signal assertion				With shared signal assertion			
		C state in P0 (MESI)	C state in P1 (MSI)	ccMC table P0	ccMC table P1	C state in P0 (MESI)	C state in P1 (MSI)	ccMC table P0	ccMC table P1
ⓐ	P0 read	**I ⇒ E**	**I**	**I ⇒ E**	**I**	**I ⇒ S**	**I**	**I ⇒ S**	**I**
ⓑ	P0 write	**E ⇒ M**	**I**	**E**	**I**	**S ⇒ M**	**I**	**S ⇒ M**	**I**
ⓒ	P1 read	**M**	**I ⇒ S**	**E ⇒ S**	**I ⇒ S**	**M ⇒ S**	**I ⇒ S**	**M ⇒ S**	**I ⇒ S**

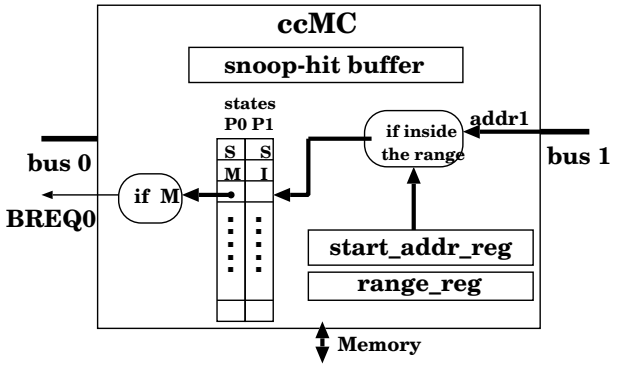

Figure 3: The bookkeeping approach

can be useful for computation-bound applications since less memory traffic will appear on the bus.

In general, the coherence protocol integrated with the bypass approach is the same as the one proposed for shared-bus architectures in [9], but with one more additional benefit. It can take advantage of all the protocol functionalities when integrating heterogeneous processors with homogeneous coherence protocols. For example, suppose that P0 has a MESI [3] protocol with the cache-to-cache transfer and P1 also uses a MESI protocol but without the cache-to-cache transfer. In this case, the integrated protocol on a shared-bus architecture will prohibit the cache-to-cache transfer, restricting the usage of all protocol states. However, the bypass approach can preserve all the states in the protocol because ccMC plays a role as a buffer between buses, and its bus interface for each side is customized to support all the functionalities that each processor provides. More details will be explained in Section 3.2.2.

The snoop-hit buffer depicted in Figure 2 is used for expediting data transfer between processors when a snoop hit occurs on a M-state cache line. Our current implementation of the snoop-hit buffer stores one cache line.

3.2 Bookkeeping approach

Figure 3 shows our bookkeeping approach. Similar to the bypass approach, the bookkeeping approach maintains one pair of registers to specify a shared memory region. A state table keeps the coherence state information for each processor. When a memory request from bus1 falls within the specified range, it records the coherence state of the memory transaction in the corresponding state table entry. Depending on the state information in the table, if it is invalid or shared,[2] ccMC does not claim the bus mastership of the other bus (bus0) and brings the data directly from main memory. As such, it eliminates unnecessary request forwardings to the other bus. For bandwidth-bound applications, the bookkeeping approach can filtrate false coherence traffic and increase the effectiveness of bus utilization. In addition, a snoop-hit buffer is also employed for performance improvement. Bookkeeping approach is more expensive than the bypass approach in terms of hardware overhead. We will analyze the hardware implementation and their cost in Section 4.

The book-keeping approach does not allow the **E**xclusive state of the coherence protocol, with an exception when integrating MEI [3] protocol with other protocols. The issue caused by the E state is illustrated in Table 1. In the example, we assume that P0 and P1 support MESI and MSI [3] protocol, respectively. There are three back-to-back memory operations ⓐ, ⓑ, and ⓒ executed on the same cache line. Operation ⓐ changes the cache line state from I to E in P0 and updates the corresponding entry in the ccMC state table. Even though the subsequent write operation ⓑ changes the state from E to M in P0, it does not appear on the bus due to a cache hit, thus no update in the corresponding entry of the ccMC table. Then, the read operation ⓒ from P1 will read a stale data from the main memory, invoking the E to S state transition in the ccMC table. These state transitions are shown from column 3 to 6 in Table 1.

To forbid the E state, we have the ccMC asserting the shared signal whenever a processor initiates a read transaction. The new state transitions with this solution is shown in the last 4 columns of Table 1. With the shared signal assertion in P0 by ccMC, operation ⓐ now changes the cache line state from I to S. Then, operation ⓑ, visible to ccMC, invokes the state changes from S to M in both the P0 and the ccMC state table. At the end, operation ⓒ receives the requested data from P0 rather than from the memory, accompanying the state changes from M to S in P0 and I to S in P1, respectively. The ccMC state table also reflects the state transitions coherently as shown in Table 1. Since the E state only accounts for a very small portion of the total state transitions, the impact of eliminating the E state is rather insignificant.[3]

Although the E state is not allowed in our book-keeping approach, the state transition to the E state is inevitable in the MEI protocol. Since the S state is absent from the MEI protocol, integrating MEI with other coherency protocols requires ccMC to employ the read-to-write conversion that eliminates the S state from the other protocols. Therefore, only the I and E states are maintained in the ccMC state table, where the I state indicates the data is not in the cache and the E state shows the data is either in the cache unmodified (true E state) or modified (hidden M state). For each request from the processor with the MEI protocol, if the ccMC state table indicates the line is in E state in the other processor, ccMC accesses the other bus and places a write operation on the bus. As such, the E state cache line is invalidated, or the M state cache line is drained out to main memory. Finally, the requester gets data either from main memory or from the snoop-hit buffer.

In the following sections, we focus on the protocol integration of our target MPMB SoC architecture. We discuss integrated protocols on the combinations of four major

this figure for brevity.

[2]If a processor supports cache-to-cache transfer mechanism in the MESI protocol, ccMC will claim the bus mastership to initiate the cache-to-cache transfer from the other processor.

[3]SPLASH2 benchmark shows only 0.76% of the time a cache line is in E state [10].

protocols: MEI, MSI, MESI, and MOESI [3]. The variations include (1) MEI with MSI/MESI/MOESI, (2) MSI with MESI/MOESI, and (3) MESI with MOESI. We further discuss SoC architectures in which both or either of the processors do(es) not have any native coherence support.

3.2.1 MEI with MSI, MESI, or MOESI

For clarity, we assume that P1 with MEI sits on bus1 and P0 with one of the following protocols MSI, MESI, or MOESI on bus0. To prohibit the transition into the S state in P0, the read-to-write conversion is employed as discussed. Therefore, ccMC claims the bus mastership of bus0 when a request by P1 is within the shared memory range and the ccMC state table indicates the E state in P0. After acquiring the bus mastership, ccMC drives a write operation on bus0 using the requested address to drain out the cache line if dirty, or invalidate the cache line otherwise. Therefore, the final integrated protocol is MEI for all three cases.

3.2.2 MSI with MESI, or MOESI

Now we assume that P1 with MSI on bus1 and P0 with other protocols (MESI or MOESI) on bus0. In these combinations, the E state is eliminated by using shared signal assertion to avoid the problem discussed in Table 1. Unlike a shared bus architecture, in a multiple bus architecture, the cache-to-cache transfer is allowed if the MESI or MOESI protocol provides it, even though a processor on the other side bus does not support the cache-to-cache transfer. It is because ccMC plays a role as a buffer between buses, and the bus interface for each side is customized to support all the functionalities each processor provides. For example, suppose that P1 with MSI requests a block of memory in the I state in its own cache, and P0 with MOESI currently has the block in the M state. Since the ccMC state table indicates the M state in P0, ccMC forwards the request to bus0. Then, P0 directly provides the block to ccMC, making the state transition from M to O in P0. ccMC buffers the block in the snoop-hit buffer, simultaneously forwarding it to bus1, which changes the state from I to S in P1. According to the specification of the MOESI protocol, main memory is not updated for this M to O transition. Main memory will be updated only when the O-state cache line is displaced. In the opposite case, when P0 with MOESI requests a block of memory in the I state in its own cache and P1 has the block in the M state, P1 changes the state from M to S, supplying the block to P0 through the snoop-hit buffer and P0 changes the state from I to S. Since the S state indicates that main memory is also up-to-date, the snoop-hit buffer has to update the main memory simultaneously while transferring the block to P0. Thus, the integrated protocol is MSI with the O state or the cache-to-cache transfer enabled

3.2.3 MESI with MOESI

Similar to the integration of MSI and MOESI, the cache-to-cache transfer is permitted in the integration of MESI and MOESI. Hence, unlike the integrated protocol on a shared bus architecture, the final integrated protocol is MESI with the O state and the cache-to-cache transfer enabled.

3.2.4 Integration with no native protocol

A normal data cache without any native coherence protocol support behaves like having the MEI protocol without any snooping capability. When a read miss occurs, a block is brought into the cache and set to valid (the E state). A subsequent write to the same line marks the block dirty (the M state). A write miss sets both the valid and dirty bits (the M state) when the line is brought in. Therefore, when integrating with other coherence protocols, the final integrated protocol is MEI. However, due to the lack of the snooping functionality, an interrupt is used to drain or invalidate a cache line, when a request is received from a processor on the other side bus.

Table 2: Hardware cost of bookkeeping approach (cache line=32 bytes)

Shared memory area	Table size (Bytes)	Synthesized result (Gates)
1KB	16B (32×4bits)	1,337
2KB	32B (64×4bits)	3,147
4KB	64B (128×4bits)	6,106
8KB	128B (256×4bits)	11,927
16KB	256B (512×4bits)	24,467
32KB	512B (1024×4bits)	50,715

4. HARDWARE COST EVALUATION

We implemented the bypass and bookkeeping approaches in ccMC using Verilog-HDL and synthesized them by Design Compiler from Synopsys with TSMC 0.18μ technology.

The bypass approach uses two comparators, two multiplexers, and two registers for each memory area with a state machine to manage the bypassing. ccMC also needs the bus master logic to drive all the buses for each processor. The synthesized result reports 356 gates.

The bookkeeping approach introduces additional cost on top of the hardware cost of the bypass approach. This additional cost mainly comes from the ccMC state table. Since the E state is not allowed as explained in Section 3, each state table entry needs two 2-bit registers for keeping three states, M, S, and I for MESI and four states, M, O, S, and I for the MOESI protocol. Table 2 shows the synthesized results for different sizes of the table covering the shared area. Note that all control logic overheads such as the table indexing are included in evaluating the design cost.

5. PERFORMANCE EVALUATION

Figure 4 shows two hardware platforms for performance evaluation. Platform ① has a PowerPC755 with MEI and an ARM core with MESI, of which the integrated protocol is MEI. Platform ② integrates an ARM with MSI and another ARM with MESI, of which the integrated protocol is MSI. We implemented the hardware platform using Verilog-HDL and Seamless [8] processor models. The Seamless CVE and ModelSim from Mentor Graphics were used for simulation. The PowerPC755 has a 32KB data cache with the MEI coherence protocol. For ARM9TDMI, we implemented an 8KB data cache with the MSI and MESI protocols using Verilog. The cache line size of both data caches is 32 bytes. The ARM core and the PowerPC processor are operated at 50MHz[4] and 100MHz respectively. The memory is synchronized at 50MHz. The cache miss penalty was varied from 14 cycles to 45 cycles for the simulations. There are two bus agents which can request bus mastership from each bus: the processor and the DMA engine. The DMA engine handles direct data transfer from peripheral devices to local memory and vice versa. It is used for injecting traffic on each bus. We assume a 100Mbps Ethernet interface on bus0 and a 320×240 resolution, 30 frames/sec LCD controller on bus1. DMA0 manages data transfer from the RX buffer to local memory and from local memory to the TX buffer while DMA1 transfers frame data for LCD display.

For the performance evaluation, the RTOS kernel simulation was performed. Simulated task insertion/deletion routines are based on a slightly modified version of the kernel routines extracted from Atalanta [12]. For inter-processor communication and synchronization, Atalanta provides a shared address space approach. Thus, processors that share system objects such as semaphores or mailboxes can access

[4]This low frequency is due to the limitation of the Seamless ARM9TMDI model, however, we expect similar simulation results at a higher frequency.

	P0 / protocol	P1/ protocol	Integrated Protocol
platform ①	PowerPC755/MEI	ARM9TDMI/MESI	MEI
platform ②	ARM9TDMI/MSI	ARM9TDMI/MESI	MSI

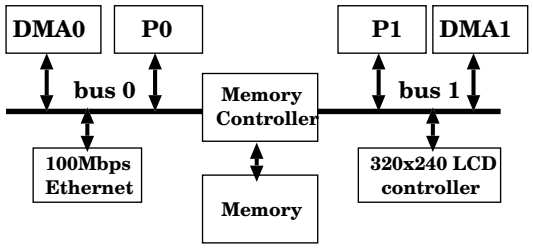

Figure 4: Simulated hardware platforms

each other's task control block (TCB). For example, assuming P0 and P1 share a semaphore S and currently P0 owns S and P1 is waiting for S. When P0 is done with the semaphore S, it releases it. Then, P0 promotes the waiting task's TCB to the ready state by changing the state field in the TCB and inserts the TCB into the ready list of P1. Then, P0 sends an interrupt to P1 so that P1 can reschedule tasks and execute the task with the highest priority. In Atalanta, the tasks' TCB on each processor is connected via a doubly-linked list based on priority. Each task's TCB has 14 word-length fields including the state field and two fields for the doubly-linked list. Atalanta also maintains an array to reference the highest-priority ready tasks, which is shared by all processors in a system. The task insertion/deletion mechanism in Atalanta RTOS has been modeled and simulated. We used the explicit software synchronization mechanism as the *baseline*.

5.1 Performance of bypass approach

Figure 5 and Figure 6 show the simulation results with 2 tasks and 32 tasks for each processor. For 2 tasks with a miss penalty of 14 cycles, platform ① shows a 2.20X speedup while platform ② shows a 1.64X speedup without the snoop-hit buffer. When the snoop-hit buffer is employed, the speedup is increased to 2.28X and 1.70X for platform ① and ② respectively. Snoop-hit buffer boosts the performance because it shortens data transfer time upon a snoop-hit in ccMC. For 32 tasks with the same miss penalty, platform ① shows 6.65X and 6.57X speedups and platform ② shows 4.78X and 4.71X speedups with and without snoop-hit buffer. Speedup differences between two platforms come from different processor cores. Since PowerPC755 is 2-way superscalar machine running twice faster than the ARM core, platform ① shows better speedup numbers than platform ②. For a lower miss penalty, the execution time of the software solution is more conspicuous since the total execution time is comparably smaller than the one with a longer memory access time. As the miss penalty increases, the total execution time largely depends on the memory access time. The rate of the execution time increase is faster in proposed approaches than in the baseline software solution since the software solution was already slow. Therefore, the speedup of a smaller miss penalty is higher than the one of a larger miss penalty and it becomes saturated as the miss penalty is large enough.

Figure 5 shows similar speedup patterns between two platforms even though the integrated protocols are different (MEI and MSI). This is a result of a much shorter length of the TCB's linked list. Since only two tasks are running on each processor, the length of the doubly-linked list is two. The insertion and deletion of the linked list demands the

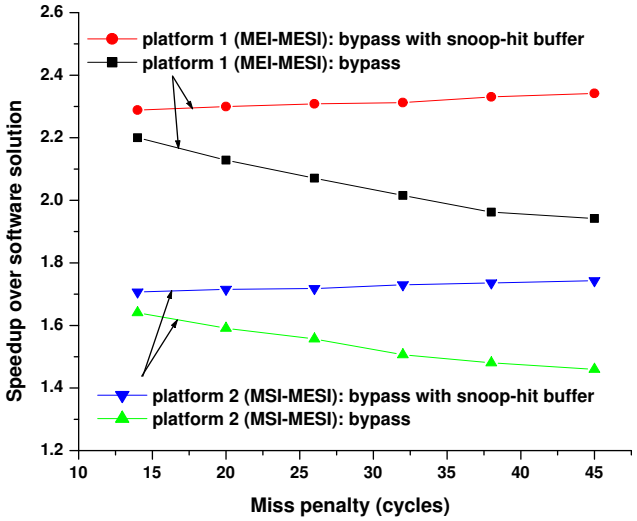

Figure 5: RTOS kernel simulation results with 2 tasks per each CPU

modification of fields in both lists, resulting in the useless S state. On the contrary, in Figure 6, the MSI protocol enjoys the benefit of the S state since there are 32 tasks on each CPU and only two or three TCBs need to be modified depending on the position of insertion and deletion. Therefore, the speedup slope in platform ② is less steep than the one in platform ①.

5.2 Performance of bookkeeping approach

In RTOS simulations, the bookkeeping approach did not show any significant performance improvement over the the bypass approach, thus we did not show them in the figures. This is due to the tiny data working set used. Note that only 56 cache lines are needed for all the TCBs of the 32 tasks. In fact, the bookkeeping approach has an advantage over the bypass approach. It happens when one processor misses a cache line and the state table indicates that the other processor either does not have the line in its cache or has the line in S state. In this case, the requesting processor can acquire this line directly from main memory instead of snooping the other processor's cache as in the bypass approach. Due to the restriction of the small working set in our RTOS simulations, this advantage was not clearly shown in our prior analysis. To illustrate this advantage, we modeled a synthetic benchmark by running one single task on each processor and having each task try to access the same memory blocks. Before entering a critical section in the shared memory, each task needs to acquire a lock. Upon exiting the critical section, all the shared blocks accessed are forcibly evicted by contrived conflict misses in the benchmark. DMAs with the cycle-steal mode are used to inject traffic on all buses like the RTOS simulations. We studied the performance sensitivity under different bus bandwidth utilizations by controlling the amount of the DMA traffic injected on the bus. For example, in our simulation results in Figure 7, the 25% bus utilization (x-axis) means that DMA0 and DMA1 both request their own bus for every 32 cycles and transfer data for 8 cycles once granted.[5] Figure 7 shows the speedups of bookkeeping over bypass as

[5]However, it does not mean that DMAs always uses 25% of the bus bandwidth since processors or snoop requests might be granted already for the bus when DMAs request the bus mastership.

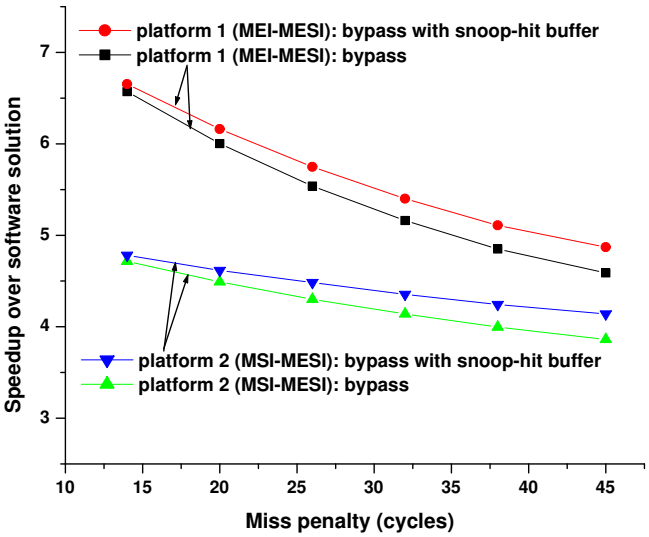

Figure 6: RTOS kernel simulation results with 32 tasks per each CPU

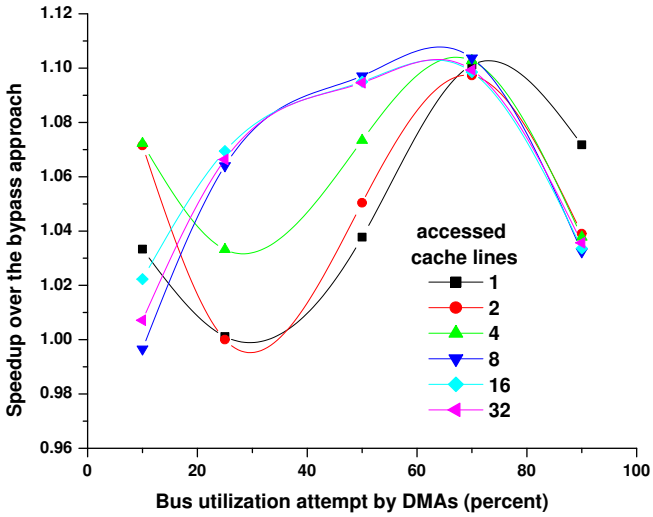

Figure 7: Simulation results for comparison between the bypass and bookkeeping approach

the bus utilization rate by DMA injection increases from 10% to 90%. As shown, the bookkeeping approach shows up to 10% performance improvement compared to the bypass approach. In general, the performance continues to increase up to the point when the bus utilization used by DMAs reaches 70%. After that, DMAs are very likely to block both the snoop requests and the processor's requests in both approaches, thus declining the speedups. We also observed some outliers cases. For example, when the number of shared cache lines is 8 and bus utilization by DMAs is 10%, the bypass approach shows a slightly better performance. By analyzing the simulation waveforms, we found that by coincidence DMA operations are synchronized and periodically delay processors' requests in the bookkeeping approach, but not in the bypass approach. Two other cases show the same behavior when the bus utilization is changed from 10% to 25% with 2 and 4 cache lines shared.

6. CONCLUSIONS

In this paper, we proposed efficient integration techniques — bypass and bookkeeping — for maintaining cache coherence among heterogeneous processors on a non-shared bus SoC architecture. A cache coherence-enforced memory controller (ccMC) is proposed to awake the snooping capability of processors on MPMB SoCs. The bypass approach is a very inexpensive solution for coherence. The bookkeeping approach filters out unnecessary forwardings, resulting in more performance improvement over the bypass approach, at the expense of additional hardware. The proposed two approaches show a significant speedup (up to 6.65X) over the naive software solution in the RTOS kernel simulations. The bookkeeping approach reports up to 10% performance improvement over the bypass approach in the synthetic benchmark simulations. As more and more SoCs in commercial market adopt heterogeneous multiprocessors with real-time constraints, we strongly believe that our methodology provides cost-effective solutions for efficient communication among heterogeneous processors.

7. REFERENCES

[1] L. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *Proc. of the Int'l Symp. on Computer Architecture*, 2000.

[2] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Proc. of the In'tl Conf. on Architectural Support of Programming Languages and Operating Systems*, pages 224–234, Apr. 1991.

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

[4] K. Gharachorloo, M. Sharma, S. Steely, and S. V. Doren. Architecture and Design of AlphaServer GS314. In *Proc. of the Int'l Conf. on Architecture Support for Programming Languages and Operating Systems*, 2000.

[5] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. rosenblum, and J. Hennessy. The Stanford FLASH Multiprocessor. In *Proc. of the Int'l Symp. on Computer Architecture*, 1994.

[6] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proc. of the Int'l Symp. on Computer Architecture*, 1997.

[7] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proc. of the Int'l Symp. on Computer Architecture*, 1990.

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

[9] T. Suh, D. M. Blough, and H.-H. S. Lee. Supporting Cache Coherence in Heterogeneous Multiprocessor Systems. In *Proc. of the Conf. on Design, Automation and Test in Europe*, 2004.

[10] T. Suh, H.-H. S. Lee, and D. M. Blough. Integrating Cache Coherence Protocols for Heterogeneous Multiprocessor Systems, Part 1. *IEEE Micro*, July/August 2004.

[11] T. Suh, H.-H. S. Lee, and D. M. Blough. Integrating Cache Coherence Protocols for Heterogeneous Multiprocessor Systems, Part 2. *IEEE Micro*, September/October 2004.

[12] D.-S. Sun, D. M. Blough, and V. Mooney. Atalanta: A New Multiprocessor RTOS Kernel for System-on-a-Chip Applications. Technical Report GIT-CC-02-19, CERCS, Georgia Institute of Technology, 2002.

[13] W. Wolf. The Future of Multiprocessor Systems-on-Chips. In *Proc. of the 42th Design Automation Conference*, 2004.

[14] S. Yoo, G. Nicolescu, D. Lyonnard, A. Baghdadi, and A. A. Jerraya. A Generic Wrapper Architecture for Multi-Processor SoC Cosimulation and Design. In *Proc. of the Int'l Symp. on Hardware/Software Codesign*, 2001.