Integrating Cache Coherence Protocols for Heterogeneous Multiprocessor Systems, Part 1

THIS SYSTEMATIC METHODOLOGY MAINTAINS CACHE COHERENCY IN A HETEROGENEOUS SHARED-MEMORY MULTIPROCESSOR SYSTEM ON A CHIP. IT WORKS WITH ANY COMBINATION OF PROCESSORS THAT SUPPORT ANY INVALIDATION-BASED PROTOCOL, AND EXPERIMENTS HAVE DEMONSTRATED UP TO A 51 PERCENT PERFORMANCE IMPROVEMENT, COMPARED TO A PURE SOFTWARE SOLUTION.

Taeweon Suh
Hsien-Hsin S. Lee
Douglas M. Blough
Georgia Institute of Technology

With advances in lithography, an entire system fits on a single chip, in what the industry calls a system on chip (SoC). Even though this ever-increasing chip capacity offers embedded-system designers much more flexibility, the requirements of a contemporary embedded system—high performance, low power, real-time constraints, and fast time-to-market—typically still restrain designs. To meet these often contradictory requirements, embedded processors continue to evolve on both the programmable general-purpose processor's side (that of ARM and MIPS) and the configurable processor's side (processors from ARC, Improv, and Tensilica). Regardless of the embedded processor's evolutionary path, the application's properties dominate the design of SoC platforms, that is processors are assigned based on the nature of the required task.[1]

General-purpose processors target control-centric and some signal processing tasks with low-to-medium performance requirement. In contrast, SoCs integrate application-specific processors such as digital signal processors (DSPs) to achieve high performance. Time-critical tasks with extremely high-performance demands, such as inverse discrete cosine transform (IDCT) and fast Fourier transform (FFT), typically require dedicated hardware. To minimize time to market, designers commonly base designs on intellectual property (IP). The heterogeneous demands of the applications naturally lead to heterogeneous multiprocessor and/or IP-based SoC platform design. This design trend is already visible; many multimedia, wireless, network, and gaming systems have arisen from just such an approach. For example, Texas Instruments' OMAP 2, LSI's DiMeNsion 8650, Analog

0272-1732/04/$20.00 © 2004 IEEE Published by the IEEE computer Society

Devices' GSM baseband processor AD6525, and Philips' Nexperia pnx8500, to name a few, are examples of such designs. Therefore, it is imperative to provide an efficient and effective design methodology to cover a heterogeneous processor system for embedded SoC design. In academia, reflecting this trend, various multiprocessor researches in embedded systems are being actively conducted.[2,3]

The design complexity of integrating heterogeneous processors on SoCs is not trivial since it introduces several problems in both design and validation because of the various bus interfaces and incompatible cache coherence protocols. In this article, we focus on cache coherence issues for heterogeneous multiprocessor SoCs. In addition, we also examine two essential components of such systems: lock mechanisms and real-time operating systems.

Related work

Large-scale heterogeneous multiprocessing systems contain distributed shared memory. Such a system can use a directory-based cache coherence[4] scheme for coherency among its distributed shared memory. Directory-based protocols address the intercluster coherence issues of a distributed shared-memory system while the bus-based snoop mechanism maintains intracluster coherence. The directory-based protocol can also address the coherence issue among homogeneous or heterogeneous clusters. Snoop-based bus protocols, however, fail to address the coherence problem for intracluster *heterogeneous* processors because of the distinct nature of each individual coherence and bus protocol.

In the embedded-SoC domain, researchers have proposed a design methodology for an application-specific multiprocessor SoC that uses the concept of a wrapper to overcome the problem of incompatible bus protocols.[5,6] Wrappers allow automatic adaptation of physical interfaces to a communication network. Generic wrapper architectures and automatic generation method can also facilitate the integration of existing components.[7]

Integrating heterogeneous processor platforms

Shared-memory multiprocessor architectures employ cache coherence protocols to guarantee data integrity and correctness when each processor shares data and caches data within itself.[8] For example, IBM's PowerPC 755 supports the MEI (modified, exclusive, and invalid) protocol, and Intel IA-32 processor family supports the MESI (modified, exclusive, shared, and invalid) protocol (http://developer.intel.com/design/pentium4/manuals/245472.htm). Modern processors use several variants of MESI, such as Sun Ultrasparc's MOESI protocol (exclusive modified, shared modified, exclusive clean, shared clean, and invalid) and the AMD 64's slightly different MOESI protocol (modified, owned, exclusive, shared, and invalid). DSPs, such as TI's TMS320C621x, C671, and C64x series also start to include snooping capabilities even though the snooping function is somewhat limited. For instance, their snooping function only maintains the coherence between the CPU and the enhanced direct memory access to the level-2 cache (http://focus.ti.com/docs/apps/catalog/resources/appnoteabstract.jhtml?abstractName=spru609b).

Like its homogeneous multiprocessor counterpart, the heterogeneous multiprocessor platform is in need of cache coherence support to enable data sharing in memory. However, because of the incompatibility of the distinct coherence protocols, designers need to use special design techniques during integration. The lock mechanism for critical sections also requires special consideration because the functionality and interface signals of the supported instructions for atomic access differ for different instruction set architectures. Finally, the real-time properties of many embedded systems demand a real-time operating system (RTOS). Because heterogeneous processors could share system objects such as semaphore, queue, and mailboxes, RTOS support for the heterogeneous environment is essential. These three primary issues—cache coherence, lock mechanisms, and RTOS for heterogenous multiprocessor systems will be discussed subsequently.

Integrating coherence protocols

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 a variant based on such a strategy as their default protocol. We also focus on them in this article.

We can classify heterogeneous processor

platforms into three classes in terms of the processors' cache coherence support, as Table 1 shows. In this table, we illustrate a dual-processor platform for brevity, although our proposed approach is easily extendible to platforms with more than two processors. PF1 and PF2 need special hardware, and the resulting coherence mechanism has a limitation, which will be discussed in Part 2. For PF3, as we will discuss shortly, simple hardware support can maintain the cache coherence.

Integrating processors with different coherence protocols restricts the usage of protocol states; only the states that the distinct protocols have in common are preservable. For example, when integrating two processors with MEI and MESI, the final coherence protocol must eliminate the S state. We present systematic integration methods that work for any combinations of invalidation-based protocols. These methods assume that the cache-to-cache sharing is implemented only in processors supporting the MOESI protocol, as most commercial processors do. The proposed methods include read-to-write conversion and/or shared-signal assertion and deassertion when integrating processors with four different major protocols: MEI, MSI, MESI, and MOESI. Furthermore, we show that a snoop-hit buffer can improve the cache coherence performance. We also propose a region-based cache coherence to use lost states of the protocols to further enhance performance.

Read-to-write conversion. Integrating the MEI protocol with others requires the removal of the shared state for coherency. 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 ⓐ, ⓑ, ⓒ, and ⓓ executed for the same cache line *C*.

Table 1. Heterogeneous platform classes.

	Cache coherence hardware	
Platform	**Processor 1**	**Processor 2**
PF1	No	No
PF2	Yes (no)	No (yes)
PF3	Yes	Yes

Operation ⓐ changes the state from I to E in P1 as a result of the read. Operation ⓑ changes the state from I to E in P2 and from E to S in P1. Since *C* is in the state E in P2, Operation ⓒ does not appear on the bus even though P1 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 P1 remains the same. Therefore, Operation ⓓ 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 snooping processors. The memory controller should see the actual operation, so it can 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, 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 an M or an E state, it writes back

Table 2. Problem and solution with MEI and MESI.

		Without proposed solution		With proposed solution	
Sequence	**Operation on cache line *C***	***C* state in P1 (MESI)**	***C* state in P2 (MEI)**	***C* state in P1 (MESI)**	***C* state in P2 (MEI)**
ⓐ	P1 reads	I → E	I	I → E	I
ⓑ	P2 reads	E → S	I → E	E → I	I → E
ⓒ	P2 writes	S (stale)	E → M	I	E → M
ⓓ	P1 reads	S (stale)	M	I → E	M → I

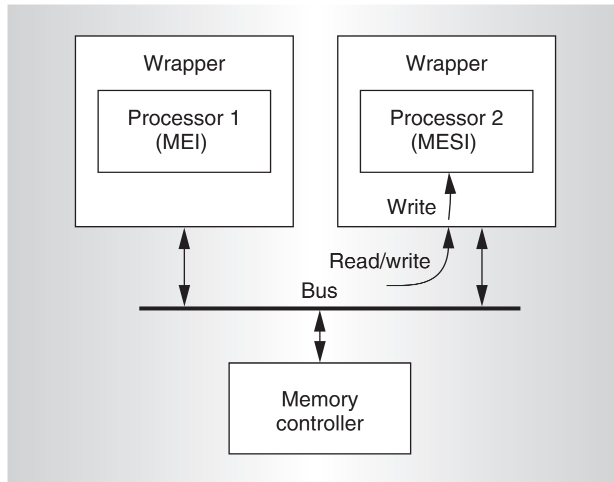

Figure 1. Method to remove the shared state.

the cache line if it is dirty and invalidates the line. This action thus excludes the shared state in the controllers' state machines. The last two columns of Table 2 illustrate the state transitions with our proposed solution. Transaction ❷ invokes the state transaction from E to I in P1, since P1 observes a write operation on the bus. Transaction ❹ changes the state from M to I in P2, since a snoop-hit on an M-state cache line causes the state change to I in the MEI protocol.

A write miss in a write-back cache initiates a bus-exclusive read (BusRdX) on a bus as explained in a later section on region-based cache coherence. In general, the way to generate the BusRdX information differs, depending on bus protocols. Thus, we use the generic signal, BusRdX, for cost estimation. The implementation requires us to assert the BusRdX signal to snooping processors within a wrapper even in a normal read by a bus master processor. We used Synopsys Design Compiler for evaluating our implementation using the 0.18-micron Taiwan Semiconductor Manufacturing Corp. library. The synthesized result shows that implementing read-to-write conversion using BusRdX requires only two gates.

Shared-signal assertion and deassertion. In integrating MSI and MESI protocols, the E state is not allowed. Suppose that P1 supports the MSI protocol, P2 supports the MESI protocol, and P1 and P2 execute the operations in Table 3 for the same cache line *C*.

Operation ❶ changes the state from I to S in P1. Operation ❷ makes the state transition from I to E in P2 because P1 cannot assert the shared signal,[4] while P1's cache line status remains unchanged. Operation ❸ invokes only the E to M transition in P2. As a result, P1 reads the stale data in ❹ because of a cache hit indicated by the S state. Therefore, the E state should not be allowed in the protocol. Our technique for removing the E state from the MESI protocol is to assert the shared signal whenever a read miss occurs. With this technique, Operation ❷ invokes the state transition from I to S in P2, and Operation ❹ changes the state from M to S in P2.

The implementation of the shared-signal assertion requires asserting the shared signal to a bus master within a wrapper whenever a read miss occurs. The synthesis result of our Verilog implementation shows that shared-signal assertion requires 1.3 gates; shared-signal deassertion shows the same result.

We discuss the details of these integration methods in an earlier work, including variations such as MEI with MSI/MESI/MOESI, MSI with MESI/MOESI, and MESI with MOESI.[9]

Snoop-hit buffer. Consider the situation where a processor initiates a read or write transaction and a snoop hit occurs in a modified

Table 3. Problem and solution with MSI and MESI.

		Without proposed solution		With proposed solution	
Sequence	**Operation on cache line *C***	***C* state in P1 (MSI)**	***C* state in P2 (MESI)**	***C* state in P1 (MSI)**	***C* state in P2 (MESI)**
❶	P1 reads	I → S	I	I → S	I
❷	P2 reads	S	I → E	S	I → S
❸	P2 writes	S (stale)	E → M	I	S → M
❹	P1 reads	S (stale)	M	I → S	M → S

block. Then, the processor that originally requested the transaction can access the block from memory after the snoop-hit processor writes the corresponding dirty block to memory. In a *homogeneous* multiprocessor platform, where the cache coherence protocol is MOESI, the cache-to-cache sharing could occur in the same situation if the cache coherence scheme supports the function.

However, for *heterogeneous* multiprocessor platforms with a different protocol combination, the cache-to-cache sharing cannot occur because of our initial assumption that it only occurs in processors supporting the MOESI protocol. Therefore, each snoop-hit requires back-to-back bursts of external memory accesses: one for the dirty write-back and one for the read request. Because these two accesses are for the same memory location, we propose a snoop-hit buffer that keeps the snoop-hit line during a write-back transaction and supplies the line to the requesting processor for the successive read. Memory is updated simultaneously with the dirty lines' buffering into the snoop-hit buffer.

This simple, additional, hardware block can improve access latency and power consumption because reads need not activate external address and data pins. Once-filled buffer data are valid until a processor encounters any of the following conditions: the next snoop-hit, a write-miss for the same line address, or a dirty-line replacement of the same line address. In a write-back cache, a write-miss appears on a bus as BusRdX, and in write-through cache, it appears as a write operation. A dirty line replacement appears as a write operation. Therefore, a line in the snoop-hit buffer becomes invalidated during the detection of a snoop-hit or write invalidation (BusRdX or write operation) on a bus, depending on the cache policies.

Figure 2 illustrates the system with a snoop-hit buffer. This block sits on a bus and has a single buffer structure that stores a single cache line. It is accessible by all subsequent read requests from other processors until the next snoop-hit or a write invalidation occurs.

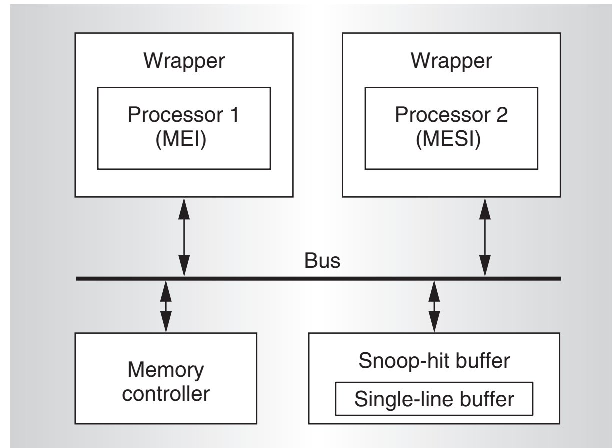

Figure 2. Snoop-hit buffer.

We can further improve performance by employing a double buffer structure. Similar to the double buffering in a video frame buffer, a double snoop-hit buffer consists of front and back buffers for keeping two cache lines. It varies from a single buffer in that the memory is not updated until the next snoop-hit of a *different* line address (sh_diff) occurs. When this happens, the logic in snoop-hit buffer copies the line in the front buffer to the back buffer. Then, the back buffer updates the memory, and the front buffer buffers a line of sh_diff simultaneously. The back buffer becomes invalidated when it finishes updating memory. However, the front buffer becomes invalidated only when detecting a write invalidation of the same line address on a bus. A double buffer can remove unnecessary memory update transactions, which could occur in the single buffer. These unnecessary transactions occur when

- other snoop-hits for the same line address before the sh_diff or
- write invalidations for the same line address before the sh_diff

follow a snoop-hit in the single buffer.

Our Verilog implementation of the snoop-hit buffer consists of a single 32-byte line buffer; a state machine for writing a snoop-hit line and reading from the snoop-hit buffer; and a memory-mapped programmable register to enable the snoop-hit buffer. The synthesized result reports 2,987 gates.

Region-based cache coherence. Even though the techniques previously described guarantee cache coherency for a heterogeneous multiprocessor environment, there is a potential performance loss caused by the lost protocol

Table 4. State transition percentages. Splash2 was executed on 16 processors, and Multiprog was executed on 8 processors.

	State transitions (percentage)	
State	Splash2	Multiprog kernel *
I	0.29	0.14
E	0.76	3.31
S	19.9	30.51
M	78.9	65.68

* data references

states, such as the S state. In Table 4, we summarize Splash2 and Multiprog simulation data from Culler, Singh, and Gupta;[10] these simulations used the MESI protocol.

As shown, the M state accounts for the majority of protocol state transitions, followed by the S state. Our integration techniques always preserve the M state, regardless of the protocol combination. However, they remove the S state in most cases of integrating different protocols. For example, our techniques remove the S state when integrating MEI with other protocols; they prohibit entrance into the S state when integrating the MESI with the MOESI.

The shortcoming of our technique is that our techniques require processors in a system to use the minimum set of the protocol states even in a situation where SoC applications share data among processors having the same protocol, which has more protocol states than the minimum set of protocols. Such a design is too restricted and prohibits compatible states, such as S, for processors using the same protocol. To address this issue, we propose a region-based cache coherence (RBCC) technique.

Given the memory area usage of the applications, region-based cache coherence conditionally permits the disabled states. Figure 3 shows a four-processor heterogeneous SoC in which three processors have the MESI protocol and one has the MEI protocol. We assume that all four processors share memory area 1 and that the three MESI protocol processors share memory area 2. We implemented the data cache of the MESI protocol using Verilog HDL and an ARM9TMDI core. The data cache has an 8-Kbyte direct-mapped structure with a 32-byte line size.

Using our RBCC technique, the new platform can use the MESI protocol for area 2 and the MEI protocol for area 1. Depending on the CPU-generated address, the extra RBCC logic will decide, on the fly, whether to enable the S state or not.

We can implement RBCC with two memory-mapped registers, one comparator, two multiplexers, and two tri-state buffers inside wrappers. Continuing with the same example, one register (start_addr_reg) keeps the starting address of area 2, and the other register (range_reg) has the range information shown in Figure 3, which also shows the shared and BusRdX signals on a bus. The shared signal informs a bus master processor that other processors have cached a line. A bus master processor uses the BusRdX signal to request an exclusive copy of a line when a write miss occurs. Most data caches are used in the write-back mode, and the write-back cache employs the write-allocation policy. So a processor uses the BusRdX to signal a write-miss and request an exclusive copy of data in write-back caches. Thus, we implement the read-to-write conversion using BusRdX inside wrappers.

Without RBCC, when integrating the MEI and MESI protocols, the BusRdX signal should always be asserted even in read misses. With RBCC, if a snoop address acknowledged by the RBCC logic falls in the range of area 2, processors view the shared and BusRdX signals on a bus through multiplexers (shown in Figure 3) that are selecting signals from a bus. Otherwise, the shared signal is deasserted (shared-signal deassertion), and BusRdX is asserted (read-to-write conversion) by multiplexers selecting 0 for the shared signal and 1 for the BusRdX signal.

The shared signal is an input when a processor is a bus master, that is, when bus GraNT (BGNT) is asserted; and it is an output when a processor is not a bus master (that is, when snooping). The BusRdX signal is an input when a processor is not a bus master and an output when a processor is a bus master. We can extend the RBCC logic in Figure 3 to include as many areas as needed. It is also easily extendable to other protocol combinations such as MEI-MOESI, MESI-MOESI, and so on.

The implementation requires components listed before for each memory area, except we need only two tri-state buffers no matter how

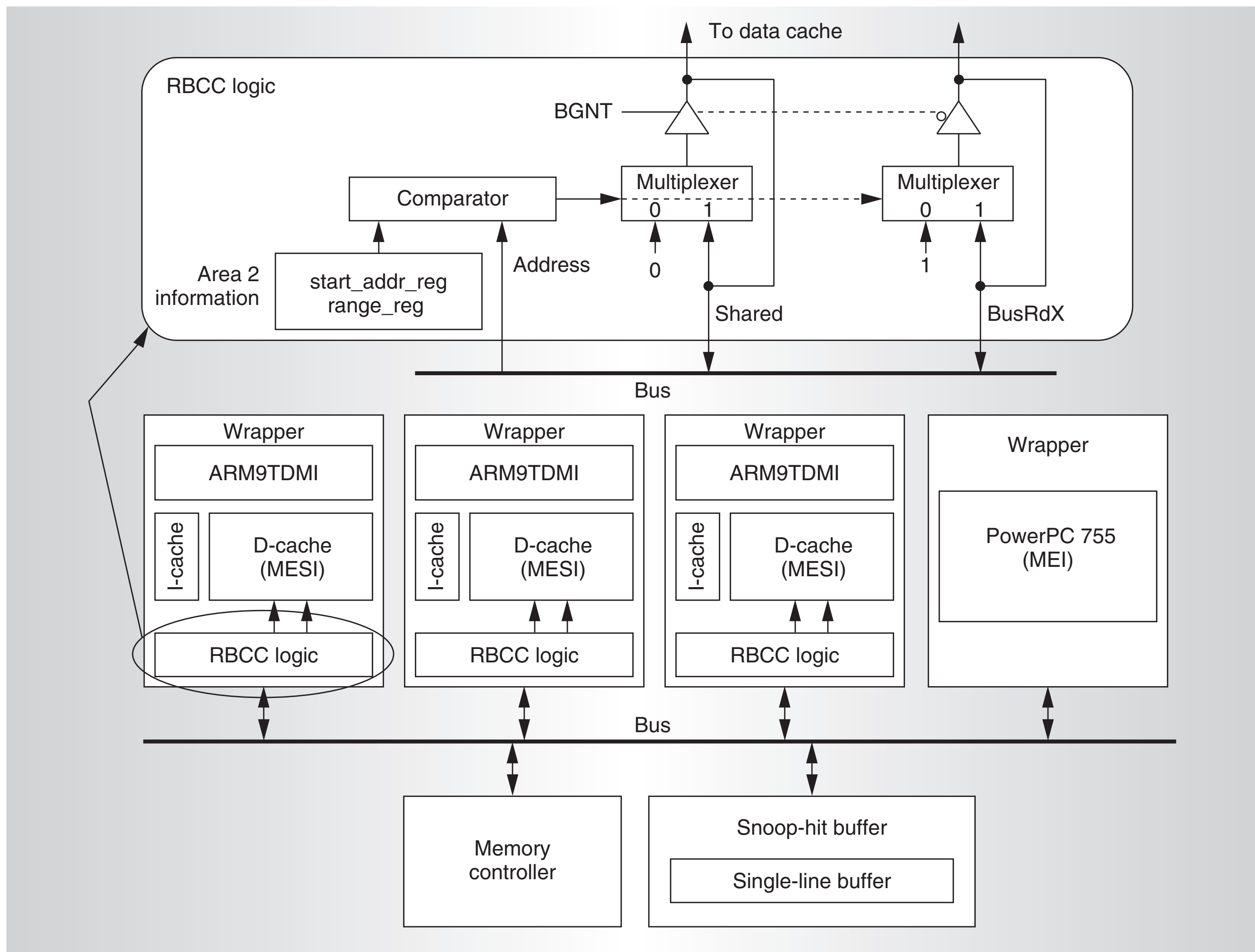

Figure 3. Region-based cache coherence on a four-processor SoC.

many areas we choose to use. Our synthesized result reports 591 gates for one memory area.

DMA. In general, designs allocate memory-mapped I/Os into uncacheable memory space. Therefore, DMA should not cause any coherence problem. However, some unconventional systems allow DMA to transfer data between cacheable regions. In these systems, we can resolve the coherence problem by allowing the DMA controller to concede bus mastership whenever a snoop-hit occurs during DMA operations and reclaim it after write back if the corresponding line is dirty. However, this DMA issue is not limited to heterogeneous platforms, but also affects homogeneous platforms.

Lock mechanism for heterogeneous platforms

In a shared-memory multiprocessor system, the processors should access critical sections in a mutually exclusive manner. To guarantee this, systems use the lock mechanism, in which processors access locks variables atomically. Processors designed for supporting multiprocessors or multithreaded systems inherently provide atomic instructions with dedicated interface signals. For example, the PowerPC755 features lwarx and stwcx instructions with RSRV signal, and the ARM processor supports swp and swpb instructions with BLOK signal.

For these instructions to work correctly, the corresponding protocol of interface signals must also have support in the memory controller to guarantee atomic accesses. Even though homogeneous platforms can take advantage of these instructions, it would be infeasible to use them in a heterogeneous multiprocessor environment

because the behavior of each atomic instruction differs for different processors. Software solutions such as the Bakery algorithm are an alternative in heterogeneous environments, but they are inefficient from a performance standpoint.

The SoC Lock Cache (SoCLC),[11] a simple yet efficient hardware mechanism, is a more attractive solution for heterogeneous environments. The SoCLC uses only a 1-bit lock register for a lock, and sits on a shared bus like our snoop-hit buffer. Because it uses general load/store instructions in acquiring/releasing a lock in an atomic fashion, a SoCLC can use the same high-level code regardless of the heterogeneity among processors.

With SoCLC, if a processor attempts to access a critical section, it first checks the lock register using a load instruction. If the lock is not in use, the lock register returns a 0 to the processor and sets the bit value to 1. Afterward, if other processors attempt to access the critical section, the lock register returns a 1 without changing its value. As such, this mechanism guarantees atomic access. Similarly, the processor releases the lock by writing a 0 to the lock register, using a general store instruction.

RTOS for heterogeneous platforms

Embedded systems, in general, require real-time properties in processing tasks, which requires the use of a real-time operating system, referred to as an RTOS, in embedded SoCs. Using an RTOS simplifies the design process by splitting the application into several tasks. To provide for real-time processing, the RTOS supports multitasking; event-driven and priority-based preemptive scheduling; priority inheritance; and intertask communications and synchronization. Especially in heterogeneous multiprocessor platforms, intertask communication and synchronization will impact system performance because processor heterogeneity could lead to inefficient shared-memory management.

Atalanta RTOS is an embedded RTOS designed at Georgia Tech.[12] For interprocessor communication and synchronization, Atalanta provides both message-passing and shared-memory approaches, whereas multiprocessor OS kernels such as Real-Time Executive for Multiprocessor Systems (RTEMS) and Operating System Embedded (OSE) rely on message passing. Therefore, in Atalanta, heterogeneous processors can share system objects such as a semaphore, mailbox, and queue by using cacheable shared memory, taking advantage of the cache coherence methodology we discussed earlier. The shared-memory approach allows much better use of shared memory, thereby increasing performance over that of a message-passing approach.[13] In addition, since mixed systems of RISCs, DSPs, and other specialized processors are assumed to be the target architectures, Atalanta's design has been tailored for heterogenous multiprocessor platforms.

We have implemented our methodologies for heterogeneous multiprocessor system design, using commercially available embedded processors. Part 2 of this article will discuss the case study of our heterogeneous multiprocessor system design, a limitation of PF1 and PF2, and the Verilog simulation results of our simple techniques, snoop-hit buffer, and RBCC. The simulation results show up to 51 percent performance improvement for low miss penalties with simple techniques, compared to a pure software solution. In RTOS kernel simulations, 77 percent performance improvement was achieved with RBCC, compared to without RBCC platform. MICRO

References

1. J.A.J. Leijten et al., "PROPHID: A Heterogeneous Multi-Processor Architecture for Multimedia," *Proc. Int'l Conf. Computer Design* (ICCD 97), IEEE CS Press, 1997, pp. 164-169.
2. A. Bechini, P. Foglia, and C.A. Prete, "Fine-Grain Design Space Exploration for a Cartographic SoC Multiprocessor," *ACM SigArch Computer Architecture News*, vol. 31, no. 1, Mar. 2003, pp. 85-92.
3. D. Sciuto et al., "Metrics for Design Space Exploration of Heterogeneous Multiprocessor Embedded Systems," *Proc. Int'l Conf. Hardware/Software Codesign and System Synthesis* (CODES 02), IEEE CS Press, 2002, pp. 55-60.
4. C.K. Tang, "Cache Design in the Tightly Coupled Multiprocessor System," *Proc. AFIPS National Computer Conf.*, IEEE CS Press, 1976, pp. 749-753.

5. S. Vercauteren, B. Lin, and H. De Man, "Constructing Application-Specific Heterogeneous Embedded Architectures from Custom HW/SW Applications," *Proc. 33rd Design Automation Conf.* (DAC 96), ACM Press, 1996, pp. 521-526.
6. S. Yoo et al., "A Generic Wrapper Architecture for Multi-Processor SoC Cosimulation and Design," *Proc. Int'l Conf. Hardware/Software Codesign and System Synthesis* (CODES 01), ACM Press, 2001, 195-200.
7. D. Lyonnard et al., "Automatic Generation of Application-Specific Architectures for Heterogeneous Multiprocessor System-on-Chip," *Proc. 38th Design Automation Conf.* (DAC 01), ACM Press, 2001, pp. 518-523.
8. M. Tomasevic and V. Milutinovic, *Tutorial on the Cache Coherency Problem in Shared-Memory Multiprocessor: Hardware Methods,* IEEE CS Press, 1993, p. 435.
9. T. Suh, D.M. Blough, and H.-H. S. Lee, "Supporting Cache Coherence in Heterogeneous Multiprocessor Systems," *Proc. Design, Automation and Test in Europe* (DATE 04), IEEE CS Press, 2004, pp. 1150-1155.
10. D.E. Culler, J.P. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann Publishers, 1999.
11. B.E.S. Akgul and V.J. Mooney, "The System-on-a-Chip Lock Cache," *Int'l J. Design Automation for Embedded Systems*, vol. 7, no. 1-2, Sept. 2002, pp. 139-174.
12. D.-S. Sun, D.M. Blough, and V.J. Mooney, *Atalanta: A New Multiprocessor RTOS Kernel for System-on-a-Chip Applications*, tech. report GIT-CC-02-19, CERCS, Georgia Institute of Technology, 2002.
13. D.-S. Sun and D. M. Blough, *Shared Address Space I/O: A Novel I/O Approach for System-on-a-Chip Networking*, tech report: GIT-CERCS-04-08, CERCS, Georgia Institute of Technology, 2004.

Taeweon Suh is a PhD student in the School of Electrical and Computer Engineering, Georgia Institute of Technology. His research interests include embedded systems, computer architecture, DSPs, and networks. Suh has a BS in electrical engineering from Korea University and an MS in electronics engineering from Seoul National University. He is a student member of ACM.

Hsien-Hsin S. Lee is an assistant professor in the School of Electrical and Computer Engineering, Georgia Institute of Technology. His research interests include microarchitecture, low-power systems, design automation, and security. Lee has a BSEE from National Tsinghua University, Taiwan, and an MSE and PhD in computer science and engineering from the University of Michigan. He is a member of ACM and IEEE.

Douglas M. Blough is a professor of electrical and computer engineering at the Georgia Institute of Technology. His research interests include parallel and distributed systems, dependability and security, and wireless ad hoc networks. Blough has a BS in electrical engineering and MS and PhD degrees in computer science from the Johns Hopkins University, Baltimore. He is a senior member of IEEE.

Send questions and comments to Taeweon Suh, Hsien-Hsin S. Lee, and Douglas M. Blough, School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, Georgia 30332-0250; {suhtw, leehs, doug.blough}@ece.gatech.edu.

For further information on this or any other computing topic, visit our Digital Library at http://www.computer.org/publications/dlib.