Symmetric Multiprocessing on Programmable Chips Made Easy

Austin Hung, William Bishop, Andrew Kennings

Symmetric Multiprocessing on Programmable Chips Made Easy *

Austin Hung ahung@uwaterloo.ca
William Bishop wdbishop@uwaterloo.ca
Andrew Kennings akennings@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada, N2L 3G1.

Abstract

Vendor-provided softcore processors often support advanced features such as caching that work well in uniprocessor or uncoupled multiprocessor architectures. However, it is a challenge to implement Symmetric Multiprocessor on a Programmable Chip (SMPoPC) systems using such processors. This paper presents an implementation of a tightly-coupled, cache-coherent symmetric multiprocessing architecture using a vendor-provided softcore processor. Experimental results show that this implementation can be achieved without invasive changes to the vendor-provided softcore processor and without degradation of the performance of the memory system.

1. Introduction

Advances in Field-Programmable Gate Array (FPGA) technologies have led to programmable devices with greater density, speed and functionality. It is possible to implement a highly complex System-on-Programmable-Chip (SoPC) using on-chip FPGA resources (e.g., DSP blocks, PLLs, RAM blocks, etc.) and vendor-provided intellectual property (IP) cores. Furthermore, it is possible to build MPoPC systems, where the number of softcore processors (e.g., Altera Nios [3] or Xilinx MicroBlaze [14]) that can be used in a MPoPC system is only limited by device resources. Previous research [13, 8, 5] investigated application-specific MPoPC systems that did not require shared memory resources. This paper focuses on general-purpose MPoPC systems that utilize cache-coherent, symmetric multiprocessing.

* This work was supported in part by a Science and Engineering Research Canada (SERC) Discovery Grant (203763-03), a grant from Altera Corporation, and an Ontario Graduate Scholarship (OGS).

We describe the design and use of general-purpose MPoPC systems based on vendor-provided IP. The systems use the familiar architecture of a shared-memory Symmetric Multiprocessing (SMP) system. The design goals are the following: (i) implementation with minimal user effort, (ii) no invasive alterations to vendor-provided IP, (iii) little if any performance penalties, and (iv) support for advanced features, namely caching.

Symmetric Multiprocessing on a Programmable Chip (SMPoPC) systems are suitable for a wide variety of embedded applications. The SMPoPC architecture offers several potential benefits including the following:

1. Enhanced embedded system performance: Processor cores may be added if device resources permit.
2. Simplified development: Time and effort can be spent on application development rather than on hardware specialization.
3. Performance improvements for computations: Many parallel algorithms exist for SMP architectures.

Our systems use 32-bit Altera Nios processors connected by an Altera Avalon Bus. This vendor-provided IP is supported by a suite of software development tools, optimized for the latest Altera FPGAs, and popular. A primary goal was to leverage the advanced features offered by softcore processors when implemented in leading-edge FPGAs. To this end, the Nios processor utilizes on-chip memory to serve as a cache to improve system performance. Unfortunately, the use of individual processor caches can introduce cache coherency problems – the Nios processor was not designed for use in tightly-coupled SMP architectures. This issue is addressed with a hybrid hardware/software solution that avoids invasive alterations to the Nios processor or the Avalon Bus. Thus, the solution serves as an "add-on" to existing Nios systems to facilitate cache coherency.

This paper is organized as follows. Section 2 provides an overview of SMP and cache coherency. Relevant details of the vendor-provided IP are provided in Section 3. A

Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'05)
1530-1591/05 $ 20.00 IEEE

design for a cache-coherent SMPoPC system is described in Section 4. Section 5 describes the experimental results obtained using several cache-coherent SMPoPC systems. Conclusions and future work are presented in Section 6.

2. Symmetric Multiprocessing

SMP systems are a subset of multiple instruction, multiple data stream parallel computer architectures [6]. Each processing element independently executes instructions on its own stream of data. SMP systems can function equally well as a single-user machine focusing on a single task with high efficiency, or as a multiprogrammed machine running multiple tasks [7]. SMP systems share memory resources using a shared bus as illustrated in Figure 1.

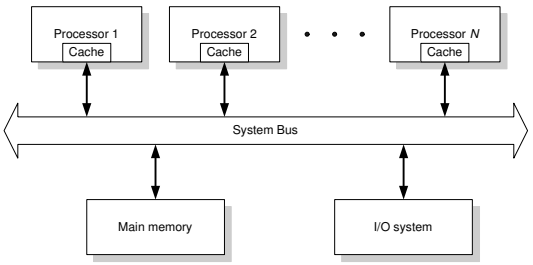

Figure 1. Typical SMP system architecture

The symmetry aspect is three-fold: (i) all processors are functionally identical and the hierarchy is flat with no master-slave or interprocessor communication relationships; (ii) all processors share the same address space; (iii) all processors share access to the same I/O subsystems with interrupts available to all processors. This symmetry helps to eliminate and reduce potential bottlenecks in critical subsystems and leads to software standardization in which system developers can produce systems with different numbers of processors that can execute the same binaries [11].

2.1. Processor Identification

SMP systems require a means to uniquely identify processors to permit selection of a bootstrap processor (BSP) for global initialization upon startup and to allow software (e.g., an operating system) to assign processes and threads to processors. Unique identification of processors is one issue addressed by our system.

2.2. Cache Coherency

Processor performance can be improved via one or more levels of *cache* located between the processor and main memory. A processor accessing cached data does not require any memory bus transactions, thereby reducing bus contention. Caches are designed to follow either a *write-through* or *write-back* policy. A write-through policy specifies that a value to be stored is written into the cache and the next level of the memory hierarchy. A write-back policy only writes to the next level of the memory hierarchy when the written cache line is replaced.

The presence of multiple, independent processor caches in a SMPoPC system leads to a cache coherency problem. Put simply, the view of the shared memory by different processors may be different. Consider a system utilizing two processors (P_1 and P_2) with local write-through caches (C_1 and C_2). When processor P_1 reads memory location X, the value is stored in cache C_1. If processor P_2 subsequently writes a different value to memory location X, cache C_1 will contain a stale value in memory location X. If processor P_1 subsequently reads location X, it will retrieve old data from the local cache. [7]

Two classes of protocols are often used to enforce cache coherency: snooping and directory. *Snooping* protocols involve having processor caches monitor (snoop) the memory bus for writes by other processors. If the local cache contains the address being written, the cache either invalidates the cache line or updates its contents. *Directory* protocols use a central directory to track the status of blocks of shared memory. When a processor writes to a shared memory block, it secures exclusive-write access to the block. Messages are passed to ensure that no stale memory blocks exist in the local processor caches. Neither of these two methods are appropriate for the SMPoPC system described in this paper as shown in Section 3 and Section 4.

3. The Nios Processor and Avalon Bus

The SMPoPC system described in this paper is based on one or more Nios processors connected to an Avalon Bus. The Nios processor and the Avalon Bus are vendor-provided IP cores designed for rapid SoPC development.

3.1. Nios Embedded Softcore Processor

The Nios embedded softcore processor is designed specifically for SoPC design. It features a 5-stage pipeline architecture and comes in both 16- and 32-bit variants. The memory system uses a modified Harvard memory architecture with separate bus masters for data and instruction memory. Memory-mapped I/O is used to access memory and peripherals attached to an Avalon bus. The global address space is configurable at system generation time. The processor can implement up to four custom user instructions. Custom logic can be placed within the arithmetic logic unit to implement single or multiple cycle instructions. Further details are found in [3].

The following Nios features are particularly relevant for this paper: (i) takes advantage of on-chip memory for caching, (ii) provides control registers for invalidating particular cache lines, (iii) supports of up to 64 vectored exceptions including interrupts generated by external hardware, and (iv) interfaces to an Avalon Bus. It is also important to note that the Nios was never intended for use in a cached SMP architecture.

3.2. Avalon Bus

The Avalon Bus is not a "traditional" bus, but rather a "switch fabric" used to interconnect processors and other devices in a Nios embedded processor system [2]. The bus provides point-to-point connections and supports multiple, simultaneous bus masters [1] (i.e., there is a dedicated connection from each potential bus master to each slave device that it can master). Although each processor and device appears to connect to a shared bus, there are no shared lines in the system. This structure is illustrated in Figure 2.

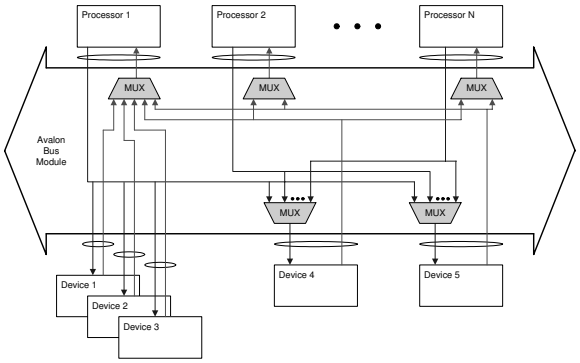

Figure 2. Structure of an Avalon Bus module

The Avalon Bus effectively prevents the use of bus snooping protocols and directory protocols to implement cache coherency. A non-trivial amount of *invasive* hardware re-development would be necessary *after* system generation to monitor every set of primary bus connections, denoted by ovals in Figure 2.

4. Design and Implementation

We now describe the design and implementation of a cache coherency module (CCM). The goal of the CCM is to enforce cache coherency with a minimum of alterations to existing vendor-provided IP. This requires a careful examination of the Nios and the Avalon Bus to understand which features facilitate and hinder cache coherency. It is also advantageous to make the process of instantiating a cache coherent SMPoPC as seamless and transparent as possible, with little if any deviation from the existing system generation procedure.

4.1. Processor Identification

The Nios processor has a `CPU ID` control register that stores the version number of the Nios processor. This register is read-only so it is not well-suited for processor identification. Several solutions exist: (i) changing the value of the `CPU ID` control register, (ii) adding a control register to the Nios, (iii) implementing a small ROM for each processor to store a unique processor ID (PID), and (iv) implementing a custom instruction in each Nios to return a unique value. The only non-invasive solution is (iii). In addition to being simple and non-invasive, this approach takes advantage of the Avalon Bus architecture to make each ROM accessible to its corresponding processor. This exclusivity also allows the ROMs to be assigned the same address to conserve address space.

4.2. Architecture

In the context of a SMP Nios system, traditional snooping or directory protocols are not suitable for ensuring cache coherency. A directory protocol would either require a dedicated bus or consume additional bandwidth on an already congested system bus. A directory protocol would require invasive changes to each Nios processor so that its cache could send, receive and understand directory protocol messages. Such a protocol would also incur a large hardware cost in the form of the central directory.

A snooping protocol would require snooping hardware to be added to monitor the caches. The Nios processor implements a pair of instruction and data caches with a write-through policy [4]. Traditionally, cache coherency is enforced by creating a hardware module for each cache that monitors the processor's memory bus. This, unfortunately, is not feasible due to the point-to-point nature of the Avalon Bus. Thus, a traditional snooping protocol cannot be used.

Our solution involves adding a slave peripheral to the system module to inform processors of memory writes. Implementing cache coherency through a slave peripheral allows system developers to instantiate the peripheral using the standard system generation tools. This solution is easy to implement since the Avalon Bus is well-specified. This is, in reality, a *hybrid snooping protocol*, that uses a centralized peripheral module to snoop the bus and maintain a "directory" to enforce coherency. The module can access the relevant signals on the Avalon Bus. This solution works with standard interfaces to peripherals (e.g., on-chip RAM, memory controller, etc.) or with special interfaces (e.g., tri-state bridge used to communicate with off-chip SRAM and flash memories).

Figure 3 shows the CCM in relation to a typical N-way SMP Nios system. The CCM must be able to detect writes as well as read the address bus. This allows the module to notify the processors of modified memory addresses, so that the appropriate cache line can be invalidated. The cache line must be invalidated, as opposed to updated, since the Nios only possesses the ability to invalidate particular cache lines. Invalidation is performed by writing the appropriate address to specific control registers implemented in each Nios processor. An update policy would require invasive changes to the system.

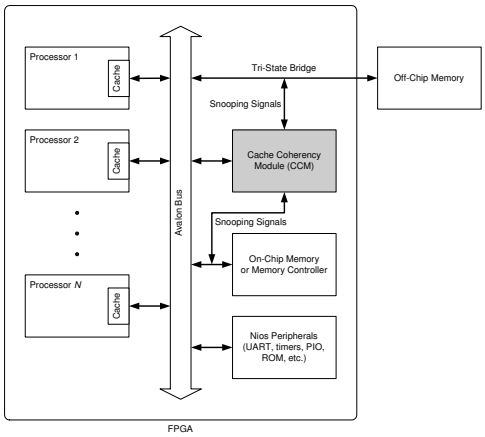

Figure 3. SMPoPC Nios system architecture

The implementation of cache clearing through processor control registers requires that software play a role in maintaining coherency. Due to the importance of maintaining coherency, the software component was written as a high-priority interrupt service routine (ISR). The CCM, a slave peripheral, raises an interrupt to enforce cache coherency.

4.3. Hardware Cache Coherency Module

The cache coherency module is divided into three components: a write detect component (one per memory device to be monitored), a central register bank component (one per system), and an Avalon slave port component (one per processor). The write detect component is responsible for snooping a single memory bus and detecting writes. For each write that is detected, the address is stored in a FIFO queue. If one or more FIFOs contain entries, then the 1-bit `STATUS` register is set, causing the Avalon slave ports to interrupt all of the processors. The 32-bit `ADDRESS` register stores the current address being processed. Once each processor in the system has read `ADDRESS`, the next address is processed. The `STATUS` register is cleared once all addresses have been processed allowing all processors to resume normal execution. Finally, a 1-bit `CONTROL` register allows the CCM module to be disabled to prevent the setting of the `STATUS` register. The CCM can be introduced into a system using the vendor-provided system generation tools without user intervention. Further details, including schematics and VHDL, can be found in [9]. This is a second-generation CCM design that improves performance and resolves a flaw in the original design [10, 9].

4.4. Interrupt Service Routine

The CCM ISR was written as a "system" ISR in Nios assembly language (as opposed to a "user" ISR based on vendor-provided C/C++ routines). The user ISR overhead (an additional 61 instructions, including 30 memory accesses) was avoided, *drastically* reducing ISR latency and run-time. Our ISR is 24 instructions long and works by disabling both caches, and then individually retrieving memory addresses to be invalidated from the CCM and invalidating them. It re-enables the cache and spin locks until the `STATUS` register is cleared (which occurs when all processors in the system have invalidated all relevant addresses). Details, including assembly code, can be found in [9].

We note three inefficiencies related to this configuration. The first is that all processors clear the specified cache line, whether it contains the data or not. This is a limitation of the Nios. The second is that the writing processor will clear its cache, even though it contains the true value, resulting in a cache miss on the next read, increasing latency. Finally, the system memory bus becomes a bottleneck when an interrupt is raised as all the processors in the system attempt to fetch the ISR code. This leads to ISR latency that scales with the number of processors since all processors must acknowledge the interrupt before normal execution can resume. This can be eliminated by duplicating ISR code in on-chip ROMs, one per processor.

The Nios supports sixty-four exception vectors. The first sixteen (0-15) vectors are unused or used for various high-priority system-level routines, including debugging and register window interrupts. Vector 16 is the highest priority exception vector available for the CCM to use. The choice of this vector is acceptable since the system-level routines do not execute under normal circumstances.

4.5. Software Requirements

The software requirements for maintaining cache coherency are simple. After initialization of each of the application processors (i.e., processors not designated as the bootstrap processor), barrier synchronization causes the application processors (APs) to wait until global initialization is complete. This is similar to the Intel SMP method of holding APs in a reset state until the bootstrap processor is finished [11]. In this case, global initialization involves

installing the ISR and enabling the CCM. The application or operating system is responsible for ensuring that the per-processor frame and stack pointers do not accidentally overwrite other processors' private memory spaces.

4.6. Memory Consistency

Memory consistency refers to the rules that a particular system follows with respect to the ordering of memory accesses (reads and writes). Defining a memory consistency model is critical to ensuring correct operation of parallel shared-memory programs.

A model in which any read to a memory location returns the value stored by the most recent write operation to that location is called the strict consistency model. The sequential consistency model relaxes the strict model, specifying that all memory accesses be serialized (they execute atomically), and that operations from a single processor appear to execute in program order [12]. This model is simple and behaves as programmers expect.

The Nios is a scalar pipelined processor that statically schedules instructions. Thus, a single processor Nios system supports strict memory consistency. The same is not true for an SMP Nios system. The cache allows a processor to write a value, followed by a read to that new value, prior to write completion (i.e., prior to all other processors invalidating their cache). This relaxation is called read-own-write-early, and is allowed by sequential consistency.

An SMP Nios system follows strict program ordering since instructions are statically scheduled. Though the Nios does not execute the instruction following a write until the memory transaction is complete, this does not include any cache invalidations by other processors. The completion of the CCM ISR indicates the completion of a write, but processors continue to execute in-flight instructions while the CCM raises an interrupt. This can lead to the situation where a write to a data value occurs, followed by a write to a synchronizing variable S. If the data that is protected by S is cached and S is not, then a processor may read the new value of S, but operate on stale data in the cache.

The program ordering problem can be solved one of two ways. The first is by placing synchronizing writes three or more instructions after the data write (causing the synchronizing write to be located either as the last instruction prior to ISR execution, or after ISR execution, guaranteeing that the data cannot be accessed prior to being invalidated). The second solution is to force such data reads to bypass the cache by using the GCC `volatile` keyword or `PFXIO` assembly instruction.

An SMP Nios system also makes writes appear atomic. Write atomicity has two requirements: (i) all processors see writes to the same location in the same order (write serialization), and (ii) a written value cannot be read by another processor unless all processors read the same value. Both conditions are satisfied by the ISR, since all processors are forced to execute it at the same time, and no read can occur prior to invalidation.

Since writes appear atomic, if program order can be enforced, then the SMP Nios system appears to follow the sequential consistency model (though this has not been formally verified). Furthermore, such a system supports the read-own-write-early relaxation. Alternatively, since the solution to the program order problem is not necessarily a desirable solution, a relaxed write-after-write (WAW) ordering can be used instead of a sequential model. A programmer can choose to use either of the two models.

5. Experimental Results

All development and testing was performed using a Nios Development Kit, Stratix Professional Edition. This kit includes an EP1S40 FPGA with 41,250 logic elements (LEs) and 3,423,744 bits of on-chip RAM. The development board includes a 50 MHz clock, 1 MB of SRAM, 8 MB of flash and other resources. Software development was performed using the GCC toolset and hardware development was performed using Quartus II 3.2SP2. Testing was conducted using 1-, 2-, 4- and 8-way SMP systems.

5.1. Cache Coherency Tests

Two software tests were conducted to validate cache coherency. The first test performs the default start-up initialization which includes enabling interrupts and initializing the cache. A single processor is arbitrarily designated as the bootstrap processor (BSP). The BSP waits until all other processors have loaded the value of a shared variable `synch` into their cache and have begun a busy-wait loop on `synch`. The BSP then performs global initialization by installing the CCM ISR and enabling the CCM module. Finally, the BSP writes to `synch` allowing all other processors to begin normal execution. A system is cache coherent if the write to `synch` is propagated to all other processors and the other processors subsequently exit their busy-wait loops.

The second test performs consecutive writes to different addresses immediately following a write to the `synch` variable (executing before the ISR). This test ensures that pipelined writes are handled correctly. A system is cache coherent if all of the written addresses are invalidated.

Succinctly stated, the system with the CCM executed both tests correctly, whereas the system without the CCM failed both tests as expected.

5.2. Device Usage and Performance Analysis

Table 1 outlines the usage statistics for the different systems compiled into the EP1S40. Each Nios was configured with 1 kB of instruction and data cache. Additional slave peripherals (8-bit UART, CCM, and so forth) were also included. From Table 1, the EP1S40 can be populated with more than an 8-way system. Only 10% of the on-chip memory bits are used, thus there are sufficient resources to increase the cache sizes to 8 kB.

Table 1. Device usage for SMPoPC systems

Nios	Logic Elements		On-Chip Memory		F_{max}
Cores	Usage	%	Usage	%	(MHz)
1	2,861	6%	46,480	1%	100.16
2	5,790	14%	92,960	2%	83.44
4	11,708	28%	185,920	5%	69.66
8	24,302	58%	371,840	10%	61.74

Logic resources required for the CCM in each system were also computed; the CCM required 50, 73.5, 83.75 and 101.125 LEs per processor for the 1-, 2-, 4- and 8-way systems, respectively. In terms of memory bits, the CCM required $4 \times 32 = 128$ bits per memory interface per processor for a full-depth FIFO.

Table 1 indicates that the system clock frequency decreases as the number of processors increases. The critical paths for the 1-, 2-, 4- and 8-way systems were identified and all multiprocessor systems were found to have similar critical paths. These paths originated from the tri-state bus arbiters to the address lines on the tri-state bus. As the number of processors increases, the lower clock rates can be attributed to the additional arbitration logic and multiplexers for connecting to the Avalon Bus. *This critical path analysis shows that the CCM is not responsible for the decrease in clock frequency* [9]. Finally, comparing each *N*-way system with a CCM to those compiled without the CCM shows negligible differences in clock frequency; variations are -3.9% to +1.8%, further confirming that the CCM is not responsible for any decrease in clock frequency.

6. Conclusions and Future Work

We have presented the development of a generic, N-way SMP system with enforced cache coherency. With respect to our original goals, the system utilizes vendor-provided IP and leverages advanced features (i.e., caching). The resulting cache coherency module is an IP block, and therefore easy to use. Experimentation indicates that the CCM has no impact on the overall clock frequency of the systems. Our hardware/software solution to the cache coherency problem is scalable and efficient in terms of hardware. Finally, the solution is applicable to other vendor-provided IP with features similar to the Nios processor and the Avalon Bus.

Future work will focus on system efficiency. A second level cache can be added to the CCM to take further advantage of on-chip memory by reducing bus contention. The CCM can also be modified to raise individual interrupts for each processor, thus avoiding unnecessary cache clearing. Finally, simple test-and-set or fetch-and-increment hardware instructions can be added to the CCM to allow processors to perform atomic synchronization.

References

[1] Altera Corp. Simultaneous Multi-Mastering with the Avalon Bus. Application Note AN-184-1.1, Altera Corp., San Jose, California, April 2002.

[2] Altera Corp., San Jose, California. *Avalon Bus Specification Reference Manual*, July 2003.

[3] Altera Corp. Nios 3.0 CPU. Data Sheet DS-NIOSCPU-2.1, Altera Corp., San Jose, California, March 2003.

[4] Altera Corp., San Jose, California. *Nios Embedded Processor 32-Bit Programmer's Reference Manual*, January 2003.

[5] M. Collin, R. Haukilahti, M. Nikitovic, and J. Adomat. SoCrates - A Multiprocessor SoC in 40 days. In *Conference on Design, Automation and Test in Europe*, Munich, Germany, March 2001.

[6] M. J. Flynn. Very High-Speed Computing Systems. In *Proceedings of the IEEE*, number 54, pages 1901–1909, December 1966.

[7] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Francisco, California, second edition, 1996.

[8] R. Hoare, S. Tung, and K. Werger. An 88-Way Multiprocessor within an FPGA with Customizable Instructions. In *Proceedings of the 18th IEEE Intl. Parallel and Distributed Processing Symposium*, page 258b, Sante Fe, New Mexico, April 2004.

[9] A. Hung. Cache Coherency for Symmetric Multiprocessor Systems on Programmable Chips. M.A.Sc. Thesis, University of Waterloo, Waterloo, August 2004.

[10] A. Hung, W. Bishop, and A. Kennings. Enabling Cache Coherency for N-Way SMP Systems on Programmable Chips. In *Proceedings of the 2004 Intl. Conference on Engineering of Reconfigurable Systems and Algorithms*, Las Vegas, Nevada, June 2004.

[11] Intel Corp., Santa Clara, California. *MultiProcessor Specification - Version 1.4*, 1997.

[12] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[13] X. Wang and S. G. Ziavras. Parallel Direct Solution of Linear Equations on FPGA-Based Machines. In *Proceedings of the 17th IEEE Intl. Parallel and Distributed Processing Symposium*, pages 113–120, Nice, France, April 2003.

[14] Xilinx Inc. MicroBlaze RISC 32-Bit Soft Processor. Product Brief, Xilinx, Inc., San Jose, California, August 2002.