Design and program multi-processor platform for high-performance embedded processing

Yijun Liu, Zhenkun Li
Faculty of Computer,
Guangdong University of Technology, Guangzhou, China
Email: {yjliu, zkli}@gdut.edu.cn

Abstract— **Modern embedded markets call for high density computing ability, making it is difficult to use just one microprocessor to meet function requirements of high-performance embedded systems. Multiple processors, including general-purpose embedded microprocessors, digital signal processors (DSPs), ASICs and FPGA hardware accelerators, are often used in these embedded systems. Not all processors in an embedded device have the same characteristics and they are asymmetric. Heterogeneous multiprocessors bring forward difficulties in both hardware and software designs. The paper addresses the issues of supporting parallelization in asymmetric multiprocessor (AMP) environment from both hardware and software sides, including cache coherence, semaphore and embedded software programming.**

Index Terms— **Asymmetric multi-processor, Symmetric multi-processor, Cache coherence, Parallelism programming, Program model**

I. Introduction

With the development of silicon technologies, embedded chips become more powerful and have more dense computing ability. Embedded processors take the place of general-purpose PC processors not only because of their low cost but also because of their low power consumption, rich functionality and high reliability. Embedded microprocessors have been used broadly in consumer electronics (such as multimedia players and gaming devices) and communication devices (such as cellular phones and personal digital assistants). However, people's pursue for high performance will never stop. Many embedded devices still call for high computing ability, making it is difficult to use just one microprocessor to satisfy the functionalities. For example, one advanced 200 MIPS ARM processor is not powerful enough to decode MPEG4 or H.264 video signals in a set top box. In many embedded systems, more than one processor is used to achieve high performance, a number of tasks run in parallel [1]. Normally, the systems have three solutions:

1) Designing software for several symmetric general-purpose embedded microprocessors (SMP) [2] [3] running in parallel;
2) One general-purpose embedded microprocessor plus programmable hardware accelerators (FPGAs), application-specific integrated circuits (ASICs), and digital signal processors (DSPs). The general-purpose cooperates with the specific hardware to improve performance;
3) Asymmetric multiprocessors (AMP) [4] [5], blending multi-core microprocessor, DSPs, FPGAs and ASICs.

The first one is a pure software solution, in which, all processors have the same characteristics and identical. In this scenario, tasks are easy to schedule and every processor can fully use their potential processing ability if there are enough tasks. Thus, 'best-effort' and good average performance can be easily achieved. However, in embedded systems, average performance and throughput are not the most important issue. Guaranteeing 'hard' real-time is of the most important. For example, in a guided missile control system, the processing must be 100% deterministic. Hard real-time is usually guaranteed by an exclusive processor or by hardware. Therefore, embedded systems should include multiple processors to enhance processing ability, and also need application-specific processor to guarantee real-time requirement. Blending general-purpose microprocessors and application-specific processors (asymmetric multiple processors) have been the way in embedded systems for years, and it will continue to be so.

The scenario brings forward difficulties in both hardware and software designs [6] [7]. In an asymmetric multiple processors platform, all the processors should communicated in an efficient way. The issues in parallel programming should be supported, such as cache coherence, semaphore and task arrangement. New embedded software program model should be studied to improve program efficiency. Since the ARM microprocessors are most-commonly used, the paper studies the solutions in ARM multiple processors background, but the principle proposed in the paper is valid in any other multiple processor environments.

The remainder of the paper is organized as follows: Section 2 describes the necessary of asymmetric multiprocessor architecture from the study of embedded processing, a common AMP architecture is presented; Section 3 addresses the hardware support for data coherence and process synchronization of AMPs. Section 4 discusses how the processors in an AMP environment cooperate and communicate with each other; Section 5 proposes a program model for ASP embedded systems; and Section 6 concludes the paper.

II. Asymmetric multi-processor architecture

With the development of silicon technology, more and more transistors can be integrated in a single chip, multi-core processors are technique trend both in embedded and PC markets. Multi-core processors have advantages in both high performance and low-power consumption. If the processing cores of a processor are identical, the processor is called a symmetric multi-processor (SMP). ARM11 MPCore^{TM} is a SMP [9] [10]. An ARM MP-Core processor can be configured to have 1-4 processor cores. Figure 1 illustrated the architecture of an ARM11 MPCore processor with 4 processor cores. The processor cores in this processor are identical, which make it easy for a single OS to schedule tasks within cores in balance.

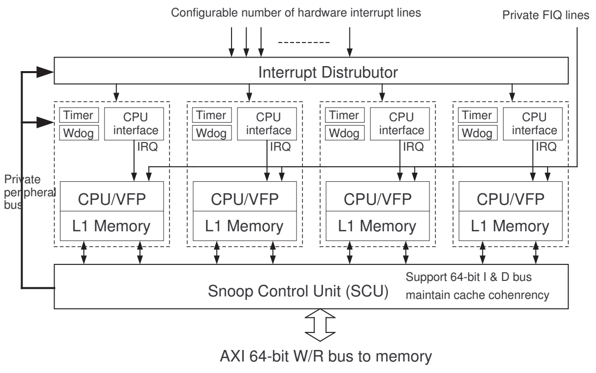

Figure 1. The Architecture of a symmetric multi-processor

As described before, SMPs usually support only pure software solutions, in which, 'hard' real-time cannot be guaranteed. The application-specific processors, such as FPGAs, DSPs and ASICs, are included to meet worst-case latency. Application-specific circuits are also involved in embedded systems to enhance the performance and power efficiency of some important program 'kernels'. In one our early paper, the characteristics of embedded programs were analyzed [8].

Embedded processors usually deal only with a fixed number of applications, and among these applications, CPU occupation rates are highly unbalanced. Embedded processors may spend most of their execution time in executing only a few loops of a few programs. Consequently, a small number of important program kernels (program segments which may be small loops or function calls) have the greatest impact on the success of an embedded processor. Moreover, since the execution and power-efficiency of these important kernels are critical for overall performance and power consumption, the kernels are normally hand-optimized, but application-specific circuits can greatly improve the efficiency of execution than pure software solutions.

Specifically designed hardware can greatly enhance the performance and power-efficiency for these important kernels, and therefore, DSPs, FPGAs and ASICs are usually selected by embedded system designers. The involvement of these application-specific processors makes the multiple processors asymmetric. The architecture of an asymmetric multi-processor (AMP) is illustrated in Figure 2.

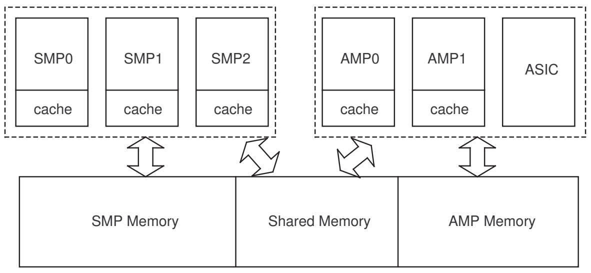

Figure 2. The Architecture of an asymmetric multi-processor

An AMP architecture includes several identical microprocessors, several different microprocessors, and some application-specific circuits. Therefore, an AMP system is a mixture of SMP and AMP. SMP and AMP have their own memory area. But they need to communicate and cooperate, a shared memory area is necessary. To overcome the bottleneck of memory and processor, a small local cache is often put between processors and main memory. A data coherence mechanism must be used to promise the correctness and consistency of data.

AMP architecture brings forward challenges for embedded hardware design and parallel software programming. The challenges are listed as follows:

- Hardware supporting parallelism:
  In AMPs, software tasks run in parallel. The parallelism must be supported by hardware, including cache coherence and atomic operations.
- Message passing and task allocation:
  The tasks running in heterogeneous multiprocessors frequently send message to each other, requiring an efficiently message passing mechanism. Moreover, tasks are loaded dynamically. Threads should be allocated to different AMP processors based on run-time conditions. An efficient task allocating mechanism is also a great challenge to AMP design.
- Software program model:
  In AMP environment, tradition embedded software program model is greatly challenged. Traditional hand-optimal programming methods depend on programmer's ability in understanding low-level hardware details and not very efficient.

We will address the solutions to these challenges in the following sections.

III. Interface module for parallelism

Cache coherence and atomic operations are necessary for parallel program execution. It is a headache and inefficient to maintain cache coherence and atomic operations in a software way. We designed an interface module for AMPs and ASICs, which automatically maintains cache coherence with SMPs and also guarantees atomic operations. The interface modules also handle message passing and dynamic task allocation. The module is put between AMPs/ASICs and buses as shown in Figure 3.

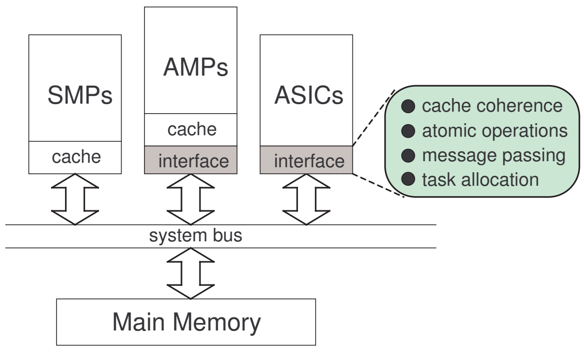

Figure 3. Interface module for parallelism

A. Cache coherence module

Cache coherence is a phenomenon existed in cached memory for multiple microprocessors [11]. A processor fetches a variable from memory and stores it in its own cache. After that, the processor modified the variable the variable in the private cache but does update the memory. Two copies of the variable exist — the one in the cache is up-to-date and the one in the memory is out-of-date. If another processor accesses the variable from the memory, it gets the invalid value. All the copies of a same variable should be consistent — this is called cache coherence.

Many cache coherence protocols are proposed. The ARM MPCore uses a write-invalidate cache protocol, called a MESI-type protocol [12]. To be compatible with ARM MPCore standard, we also design a MESI-type cache coherence module.

Using a MESI protocol, a cache line has four states:

- Modified (M states) — indicates that the processor owns the modified data block exclusively in its cache, and no other copies of the same memory location exist. Its contents are not up to date with main memory.
- Exclusive (E states) — indicates that the cached block has not been modified, and that no other copies of the same memory location exist.
- Shared (S states) — indicates that the coherent cache line is present in the cache and up to date with main memory.
- Invalid (I states) — indicates that a data block in the cache is not up-to-date and invalid.

The state of each cache line block changes depending on the read/write actions taken by the CPU. The diagram shown in Figure 4 illustrates state transfer actions of a cache line using a MESI protocol. In the figure, a solid line in the state transition indicates that the transaction is initiated by its own processor; a dotted line indicates that the state transition is initiated by a remote processor. The notation A/B means that transaction B takes place after the observation of transaction A. The meanings of

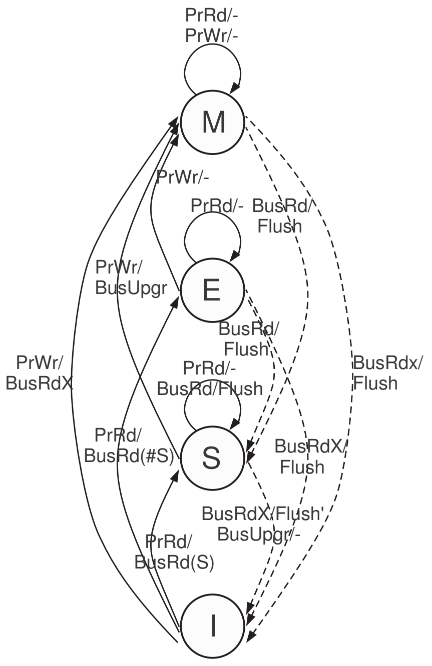

Figure 4. A MESI cache protocol

the actions are listed as follows:

- PrWr — Local processor's write operation
- PrRd — Local processor's read operation
- BusRd — Read transaction on the bus
- BusRdX — Read exclusive transaction on the bus for the ownership of a memory block
- BusUpgr — Same as BusRdX, but no data involved since the purpose is to invalidate the same memory block in remote caches
- BusRd(S) — Read transaction on the bus, and the shared signal is asserted by remote processor(s)
- BusRd(#S) — Read transaction on the bus, and the shared signal is de-asserted by remote processor(s)
- Flush — Cache line data supply to the bus for cache-to-cache transfer
- Flush' — Same as Flush, but data is supplied by the cache responsible for supplying the data
- '–' — No action taken

Based on the state machine in Figure 4, we designed a very tight cache coherence module to maintain the cache coherence of AMP and SMP. ASICs normally contain no cache; they directly read/write main memory. But they need to know if a variable is up-to-date. The data coherence module is simpler in ASICs than that in AMPs.

B. Atomic operation support

A standard problem in multiprocessor system that share data memory is to control accesses to the shared data to ensure deterministic behavior. One processor can access to the shared data areas only after another processor finishing a whole operating process of the data inside. Otherwise, you don't know which step of the process the formal processor has done, incurring indeterminate behaviors. Such sensitive data areas should be mutually exclusively accessed. Mutually exclusive access is done by some ‘semaphores’. Semaphore operations must be uninterruptible. In another word, the instructions should be ‘atomic’. In old ARM processors, the ‘SWAP’ instruction is such an atomic instruction [13]. A register is set to a ‘busy’ value, then this register is swapped with a semaphore memory location containing the Boolean. If the loaded value is ‘free’, the process can continue; otherwise, it means another processor is occupying the memory area, and the process repeats the test until it get the ‘free’ result. The SWAP operation is atomic. No other instruction can interrupt the swap operation between register and memory. The SWAP instruction locks external bus until operation finishes. This instruction is not acceptable for multiple processors because one processor could hold the entire bus until completion, disallowing all other processors. ARM MPCore introduces two new instructions — load-exclusive LDREX and store-exclusive STREX [14]. LDREX and STREX can be used for semaphore, such as the code listed below:

```
        MOV r1, #0x1
try
        LDREX r0, [LockAddr]
        CMP r0, #0
        STREXEQ r0, r1, [LockAddr]
        CMPEQ r0, #0
        BNE try
        ....
```

The lock value is put in LockAddr. The process keeps on claiming the lock until it obtains the lock. LDREX and STREX take the advantage of an exclusive monitor in the memory.

- LDREX loads data from memory:
  - If the physical address has the Shared TLB attribute, LDREX tags the physical address as exclusive access for the current processor, and clears any exclusive access tag for this processor for any other physical address.
  - Otherwise, it tags the fact that the executing processor has an outstanding tagged physical address.
- STREX performs a conditional store to memory. The conditions are as follows:
  - If the physical address does not have the Shared TLB attribute, and the executing processor has an outstanding tagged physical address, the store takes place, the tag is cleared, and the value 0 is returned.
  - If the physical address does not have the Shared TLB attribute, and the executing processor does not have an outstanding tagged physical address, the store does not take place, and the value 1 is returned.
  - If the physical address has the Shared TLB attribute, and the physical address is tagged as exclusive access for the executing processor, the store takes place, the tag is cleared, and the value 0 is returned.
  - If the physical address has the Shared TLB attribute, and the physical address is not tagged as exclusive access for the executing processor, the store does not take place, and the value 1 is returned.

Using LDREX and STREX to implement semaphore is better than SWAP because the instruction will not lock systems, granting other processors or threads access to the main memory.

The atomic instructions, such as LDREX and STREX need the support from hardware. Embedded ASPs and ASICs do not need many semaphores. We put 5 control registers to store the 5 latest exclusive monitors. A ‘snoop’ circuit is design to check the instructions in I-bus. If it finds a LDREX instruction, the address of the monitors is store in a control register. If the ASPs or ASICs want to write data to memory, they will check the 5 control registers to see whether the memory address is locked. By doing this, ASPs or ASICs will not interrupt the atomic operations of SMPs. To prevent SMPs from interrupting the atomic operations of ASPs or ASICs, we designed a circuit that can emulate the signals as LDREX and STREX. When an ASP or ASIC enters a sensitive area, it use LDREX to generate the semaphore.

IV. Coupling mechanisms

The previous section addresses the issues of data coherence and process synchronization of AMPs. This section discusses how the processors in an AMP environment cooperate and communicate with each other.

In an AMP system, a number of tasks (some tasks are software threads running on microprocessors, others are hardware executions in FPGAs and ASICs) run in parallel and cooperate with each other. At the time when system restarts, not all tasks exist. Tasks may finish and new tasks are created. Tasks may create other tasks. A software thread may create children threads. A software thread can also dynamically configure a FPGA and cooperates with the FPGA implementation (a hardware task) during runtime. During cooperation, the thread and the FPGA need to send messages to each other and synchronizations are required.

A. Message sending mechanism

Figure 5 illustrates massages sent among tasks in different processors. The message sending modes can be put in three categories:

1) Messages are sent between two symmetric processors in ARM MPCore.
2) Messages are sent between a SMP processor and an AMP processor.
3) Messages are sent between a processor and a FPGA/ASIC.

An OS is usually used in either SMP or ASP, such as Linux, eT-Kernel [15], ThreadX, etc. In the first

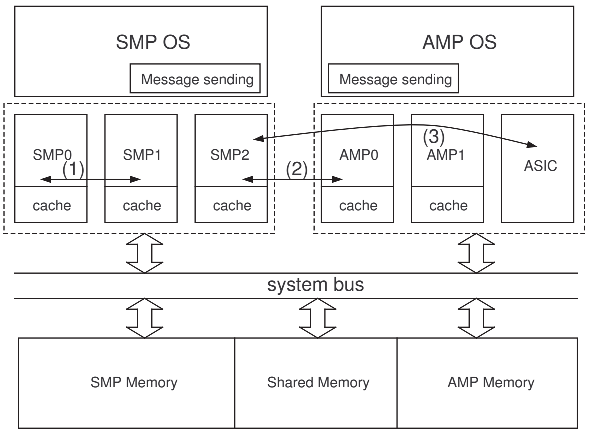

Figure 5. Message passing between AMPs

scenario, communications between SMP CPUs are best accomplished without accessing main memory. In the second scenario, communications between SMP CPUs and AMP CPUs are accomplished through an interprocessor interrupts (IPI) mechanism. The IPI typically uses an interrupt system designed to interface interrupts from I/O peripherals rather than another CPU. In these two scenarios, message passing among software tasks can be done using message sending APIs (such as POSIX thread [16]) provided by operating systems.

The emphasis of the work is message sending mechanism in the third scenario. If a message sending behavior is initiated by a FPGA/ASIC, the circuit will trigger an interrupt of processor. Otherwise, if a message sending behavior is initiated by a processor, communication between the processor and circuit is done in a way of processor-coprocessor. The circuits are designed to be compatible with ARM coprocessor standard. If processor wants to send a message to the circuit, it issues a coprocessor instruction on system bus. The 'snoop' module in the interface circuit detects the instruction and interprets it. If only small numbers of data are needed to send, the data can be sent using the several coprocessor instructions. If a big data block is needed to send, a shared memory mechanism is more efficient. Processor sends one or several coprocessor instructions to circuit. The instructions contain not data but memory location of the data block. Data fetching is done by circuit.

B. Dynamic allocation mechanism

In mixed AMPs and FPGA systems, dynamic tasks allocation means not only software thread creating/allocating but also dynamic FPGA configuration. SMP and AMP OS can handle software thread dynamic allocation automatically, but dynamic FPGA configuration is hardware-related and need to be mentioned exclusively.

A FPGA configuration bit stream is previously generated and store in a consecutive memory area. It is a very inefficient way for a processor to configure the FPGA bit by bit through coprocessor instructions as described before. Alternatively, processor only sends the start address and the size of configuration bit stream through coprocessor instructions. The 'snoop' module in the interface of FPGA detects the instruction and interprets it, then a DMA will do the configuration.

V. PROGRAMMING MODEL

Section III and IV address how to support parallel program from hardware side. This section discusses the issues of software programming environment and model for mixed SMP, AMP and FPGA systems.

The heterogeneous processing units, including symmetric processors, asymmetric processors, DSPs, FPGAs and ASICs, are integrated in high-end embedded systems, making software programming very difficult. A good software programming environment and model can cover low-level hardware details, thus greatly shortening time-to-market and increasing design efficiency.

Figure 6 shows the software architecture. User applications are defined by software designers using high-level language (as C++ and JAVA). The application is partitioned into many parallel tasks. Tasks dynamically create and destroy. Tasks run on parallel OS and communicate through communication and synchronization APIs (C/S APIs). Operating system also contains some Hardware-dependent Software (HdS) which deal with specific hardware. System software designer should modify parallel operating system (eg, a POSIX compliant operating system) for specific hardware platform and design HdS.

A good Hardware-dependent Software (HdS) allows efficient hardware-software cooperate and communication and thus to minimize performance penalties. Platform designers of embedded AMP systems are fully aware of architecture specific resource constrains, such as processor processing abilities, inter-processor latencies, available bus throughput and bandwidth, I/O speeds, etc. HdS designers should fully uses the information and send the information to OS. This will help OS optimize task allocation and gain good performance.

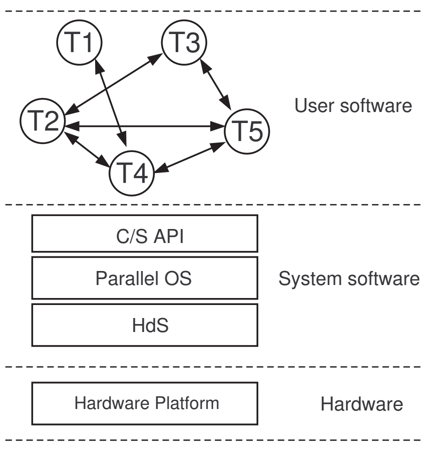

Figure 6. Software architecture

User software designer should not know many hardware details and good program environment will cover hardware details. Figure 7 show our embedded software design environment. Application programmers first write program in a traditional parallel programming way. In this step, programming is hardware independent. Programmers need to specify parallelism of processes and their interactions.

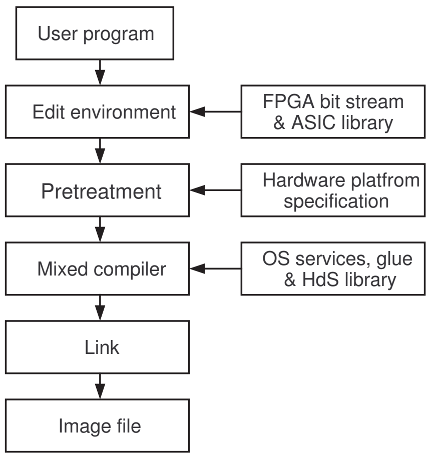
User program
Edit environment
FPGA bit stream & ASIC library
Pretreatment
Hardware platfrom specification
Mixed compiler
OS services, glue & HdS library
Link
Image file

Figure 7. Software programming environment

Designers then use a graphics-based editor to include hardware-dependent information. For example, designers can click a button to include a bit stream of a FPGA and specify when FPGA configuration should happen. The actions in the editor are recorded in specific macros. The following segment shows an example of macros.

```
C=A+B;
  /*hardware independent instructions*/
FPGAload(F3, DESbs);
  /*load DES bit file to F3 FPGA*/
SendMem(F2, v1);
  /*send variable V1 to F2*/
......
```

In pretreatment step, the micros are interpreted using real hardware parameters. In the last example — *FPGAload(F3, DESbs)*, the physic address of F3 FPGA configuration port and the size of DESbs are added.

After the processes of compile and link, the final binary image file is generated, and the binary image can be put in the memory for execution. As shown in Figure 8, the binary image is put in the main memory. Operating system creates a main process. The main process will create other processes. The processes are dynamically allocated to SMP processors and AMP processor by operating systems. Some process may incur the configuration of FPGAs. SMPs, AMPs, DSPs, FPGAs and ASICs can communicate and cooperate with each other through a message sending mechanism. The message sending mechanism is implemented through hardware.

Following the program model, application programmers can use their familiar programming methods without knowing many low-level hardware detail.

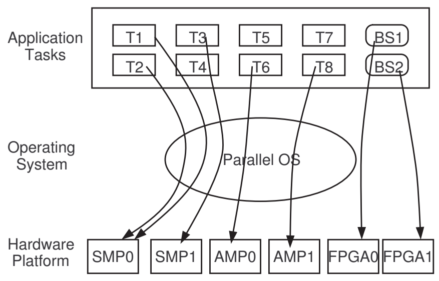

Figure 8. The execution of processes

VI. Conclusions

With the development of information technology, more and more embedded systems, such as cellular phones, personal digital assistants (PDA) and portable multi-media players, call for high processing capability, making it's difficult for a single processor to meet their performance requirements. These high-performance embedded designs usually include multiple asymmetric processors (AMP), including general-purpose microprocessors, digital signal processors (DSPs) and custom-designed FPGA hardware accelerators and Application-Specific Integrated Circuits (ASICs). AMPs greatly challenge traditional embedded system design in both hardware and software. Cache coherence and atomic operations need hardware to support. Message sending and task allocation can be done in a pure software way, but hardware can greatly increase the efficiency of message send and task allocation.

In the paper, an interface module supporting parallelism is proposed. The module is used as an interface to AXI bus for FPGAs and ASICs. The module handles cache coherence, atomic operations, message sending and task allocation in a hardware way. The paper also proposes an embedded system program model for asymmetric multiple processor systems. Following the program model, hardware details are covered. Application programmers can use their familiar programming methods with a little modification.

Acknowledgment

The project was funded by a grant from Natural Science Foundation of Guangdong (No. 07300421) and a PhD grant from Guangdong University of Technology. The authors would like to acknowledge with gratitude these grants.

References

[1] W. Wolf, The future of multiprocessor systems-on-chips, in: Design Automation Conference, 41st Conference on (DAC'04), 2004, pp. 681-685.

[2] Yingmin Li, K. Skadron, D. Brooks, Zhigang Hu, Performance, energy, and thermal considerations for SMT and CMP architectures, 11th International Symposium on High-Performance Computer Architecture, 2005. HPCA-11., 12-16 Feb. 2005 Page(s):71 - 82

[3] J. Moses, K. Aisopos, A. Jaleel, R. Iyer, R. Illikkal, D. Newell, S. Makineni, CMPSched$im: Evaluating OS/CMP interaction on shared cache management, IEEE International Symposium on Performance Analysis of Systems and Software, 2009. ISPASS 2009. 26-28 April 2009 Page(s):113 - 122

[4] A. Beric, R. Sethuraman, C.A. Pinto, H. Peters, G. Veldman, P. van de Haar, M. Duranton, Heterogeneous multiprocessor for high definition video, International Conference on Consumer Electronics, 2006. ICCE '06. 2006 Digest of Technical Papers. 7-11 Jan. 2006 Page(s):401 - 402

[5] J. M. Paul, D. E. Thomas, A. Bobrek, Scenario-oriented design for single-chip heterogeneous multiprocessors, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Volume 14, Issue 8, Aug. 2006 Page(s):868 - 880

[6] B. Senouci, A. M. Kouadri M, F. Rousseau, F. Petrot, Multi-CPU/FPGA Platform Based Heterogeneous Multiprocessor Prototyping: New Challenges for Embedded Software Designers The 19th IEEE/IFIP International Symposium on Rapid System Prototyping, 2008. RSP '08. 2-5 June 2008 Page(s):41 - 47

[7] D. Andrews, D. Niehaus, R. Jidin, M. Finley, W. Peck, M. Frisbie, J. Ortiz, P. Ashenden, Programming models for hybrid FPGA-CPU computational components: a missing link, Micro, IEEE, Volume 24, Issue 4, July-Aug. 2004 Page(s):42 - 53

[8] Y. Liu, S. Furber and Z. Li, The design of a dataflow coprocessor for power-efficient embedded processing, Proceedings of 2006 PATMOS, Springer Lecture Notes in Computer Science, September. 2006, Page: 425-438, Volume 4148/2006, ISBN 978-3-540-39094-7, ISSN 0302-9743

[9] ARM11 MPCore Processor Technical Reference Manual, ARM Limited, Lit.-Nr.:ARM DDI 0360D, 2006.

[10] Core Tile for ARM11 MPCore User Guide, Ref: DUI 0318C, 2006.

[11] M. M. Michael, A. K. Nanda, B. -H. Lim, M. L. Scott, Coherence Controller Architectures for SMP-Based CC-NUMA Multiprocessors, Proceedings of the 24th Annual International Symposium on Computer Architecture, pp. 219-228, 1997.

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

[13] Furber, S.B., ARM System Architecture, Addison Wesley Longman, 1996. ISBN 0-201-40352-8.

[14] J. Goodacre, A. N. Sloss, Parallelism and the ARM instruction set architecture, Computer, July 2005, Volume: 38, Issue: 7, On page(s): 42- 50, ISSN: 0018-9162

[15] Masaki Gondo, Blending asymmetric and symmetric multiprocessing with a single OS on ARM11 MPCore, Information Quarterly, Volume 5, Number4, 2007

[16] B. Senouci, A. Bouchhima, F. Rousseau, F. Petrot, A. Jerraya, Fast Prototyping of POSIX Based Applications on a Multiprocessor SoC Architecture: "Hardware-Dependent Software Oriented Approach", Seventeenth IEEE International Workshop on Rapid System Prototyping, 2006. Volume , Issue , 14-16 June 2006 Page(s):69 - 75

Yijun Liu was born in Jiangxi, China. He received his PhD degree from the University of Manchester, UK, in 2005, his M. Phil degree from the University of Manchester in 2003, his MS Degree from Guangdong University of Technology, China, in 2002, and his BS degree from the Beijing Normal University, China, in 1999. All degrees are in Computer Science.

He is currently an associate professor of Computer Science at Guangdong University of Technology, China. His current research interests include low-power circuit design, asynchronous logic design, computer architecture and embedded systems.

Professor Liu is a member of Computer Architecture Technical Committee, China Computer Federation.

Zhenkun Li was born in Guangdong, China. He received his BS degree in automation from the Guangdong University of Technology, China, in 1976.

He is currently a Professor of Computer Science at Guangdong University of Technology, China. His current research interests include computer architecture and embedded systems.

Professor Li is a recipient of the Chinese Government Allowance granted by the State Council.