http://ansinet.com/itj

ITJ

ISSN 1812-5638

INFORMATION TECHNOLOGY JOURNAL

ANSInet
Asian Network for Scientific Information
308 Lasani Town, Sargodha Road, Faisalabad - Pakistan

Information Technology Journal 5 (1): 192-197, 2006
ISSN 1812-5638

SPE Architecture for Concurrent Execution OS Kernel and User Code

[1]Haroon Muneer and [2]Khalid Rashid
[1]Department of Computer Sciences, International Islamic University, Islamabad, Pakistan
[2]Faculty of Applied Sciences, International Islamic University, Islamabad, Pakistan

Abstract: Operating system performance is not improving at the same rate as the speed of the execution hardware. As a consequence Operating systems are not keeping up with the demands placed on them. Computational speed up due to the increase in processor clock frequency is reaching its limits as well. Chip Multiprocessors are now being investigated to harness the silicon resources now available due to process improvements in C hip manufacturing. This research presents the study into a specialized Chip Multiprocessor for Simultaneous execution of OS kernel and user Code. SPE or Simultaneous Process Execution Architecture allows for continuous execution of OS kernel and user processes.

Key words: OS, kernel, CMP, FPGA

INTRODUCTION

As modern applications become increasingly dependent on multimedia, graphics and data movement, they are spending an increasing fraction of their execution time in the operating system kernel. Web servers have been shown to spend over 85% of their CPU cycles running operating system code[1]. For server-based environments, the operating system is a crucial component of the workload.

Multi-server and component-based operating systems are promising architectural approaches for handling the ever increasing complexity of operating systems. Components or servers (and clients) communicate with each other through cross-domain method invocations. Such interface method invocations, if crossing protection boundaries, are typically implemented through the Inter-process Communication (IPC) mechanisms offered by a microkernel. Therefore, component interaction in such systems has to be highly efficient.

Thus Inter-process Communication (IPC) by message passing is one of the central paradigms of most u-kernel based and other client/server architectures. IPC performance is vital for modern operating systems; especially u-kernel based ones. Since context and user/kernel mode switches are central to IPC operation, reducing them is a critical factor in IPC performance improvement.

A need exists to design a microprocessor that helps to improve OS performance. Specifically by improving IPC performance alone, OS performance can be improved significantly. User/kernel mode switches or context switches are a key operation in IPC. By reducing or by eliminating this context switch IPC performance can be improved significantly.

The SPE architecture platform combines several aspects of existing processor systems. The actual execution of instructions on specialized hardware originates from the earliest co-processor concept in Intel's 8086/8087 processor system [2]. The use of a configurable set of co-processors is a small step towards increasing performance by adding more specialized hardware and has been applied for numerous processor systems. In these architecture platforms, there is one master processor coordinating the activities performed by the co-processors. An example is the TriMedia architecture platform[3] originally developed Philips, which includes a single (VLIW) master processor exploiting instruction level parallelism.

SoC architecture platforms that allow true parallel execution of tasks on a number of independent master processors are also referred to as single-chip multi-processors[6]. The main design issues for such systems emerge from the necessity of communicating information between tasks running on different processors.

There are many academic and commercial CMP in existence today. Specifically the architecture discussed by Theelen and Verschueren[4] is an excellent design example. The MiP architecture platform[5] exploits parallelism at the task level.

For more efficient utilization the offered processing power could also be obtained through a higher integration of software. Although, this approach can reduce overhead and thus increase performance, it may restrict the adaptability for a wide range of products.

Corresponding Author: Haroon Muneer, Department of Computer Sciences, International Islamic University, Islamabad, Pakistan

Contacting slave processors for performing application dedicated operations requires fast on-chip interconnects. On-chip interconnects have become a very important design issue for many SoCs. The challenge is to reduce latencies for exchanging information between units that are located relatively far apart. The on-chip interconnects of the SPE architecture platform can not be compared to packet-based routing devices: because the onchip interconnect is shared stack based and allows direct function calls or OS trap invocations on the connected processors.

Efficiently accessing (off chip) memory has also been a design issue for many years. Similar to other processor systems, the SPE architecture platform uses caches to absorb memory access latencies. Main difference with other multi-processor systems is the absence of data memory for the individual master processors. With one shared memory, much programming flexibility is offered to the user.

The SPE architecture platform exploits parallelism at the task level by incorporating an independent master processor and a number of slave processors. Running multiple tasks in parallel requires sophisticated facilities for inter task communication. The architecture platform considered by Theelen and Verschueren[4] prescribes the use of so-called wrapper units to allow communication between processors. In SPE architecture platform, like describe by Theelen and Verschueren[4], communication between tasks is enabled through the use of communication resources offered by an OS kernel implemented on the master processor.

The main contribution of this study lies in the overall concept of the SPE architecture platform and more specific in the integration of a Master processor specifically designed for OS functionality.

Operating system problems: A u-kernel can provide higher layers with a minimal set of appropriate abstractions that are flexible enough to allow implementation of arbitrary operating systems and allow exploitation of a wide range of hardware.

Similar to optimizing code generators, u-kernels must be constructed per processor and are inherently not portable. Basic implementation decisions, most algorithms and data structures inside a u-kernel are processor dependent. Their design must be guided by performance prediction and analysis. Besides inappropriate basic abstractions, the most frequent mistakes come from insufficient understanding of the combined hardware-software system or inefficient implementation.

For these reasons Operating Systems have been known to cause the following set of problems:

- Operating systems are huge programs that can overwhelm the cache and TLB due to code and data size, thereby causing severe performance penalty for User programs.
- Operating systems may impact branch prediction performance, because of frequent branches and infrequent loops.
- OS execution is often brief and intermittent, invoked by interrupts, exceptions, or system calls and can cause the replacement of useful cache, TLB and branch prediction state for little or no benefit.
- The OS may perform spin-waiting, explicit cache/TLB invalidation and other operations not common in user-mode code, again effecting user code.
- In current modularized kernels, every kernel invocation causes context switch and in case of μ-kernels every call means multiple context switches, thus wasting a considerable time in switching processes.
- IPC-performance problems result from 64 bit architectures with there large number of registers and register stack engines. The large number of registers contributes to a potentially massive context (more than 2 KB) to be stored on each thread context switch[7].
- Overall, operating system code causes poor instruction throughput on a superscalar microprocessor.

To overcome these problems many techniques have been used, but each had its disadvantages. Amdahl's law tells us that if we want modern applications to run quickly, the operating system must run quickly as well. Since traditional performance models essentially ignore the operating system and modern OS-dependent applications, a need has arisen for new designs and methodologies that direct their attention at the performance of the OS kernel[1].

As mentioned earlier OS kernel workload has significantly increased, especially server based applications are putting heavy loads on the kernel. What must be realised is that we have a huge potential for performance improvement. If some how the kernel runs on an independent processor and the user code runs on another, without any bus latencies, this master-slave processor architecture can improve performance significantly. Therefore we have designed a new CMP architecture that is specifically designed to overcome OS problems.

Microprocessor architecture: Internally microprocessors have limited support for operating systems besides the features that are critical for current protected virtual memory based operating systems, like μ-kernels base

operating systems. As we have seen that modern applications are spending an increasing fraction of their execution time in the Operating System (OS) kernel.

At the multi-processor level performance improvements are due to SMP, NUMA or clustering. In each of these techniques the processing nodes are either running a copy of the kernel or the whole OS. None of these are aimed at improving OS performance directly. Rather the earlier mentioned OS problems appear at each node.

Integrated circuit processing technology offers increasing integration density, which fuels microprocessor performance growth. It is becoming possible to integrate a billion transistors on a reasonably sized silicon chip. At this integration level, it is necessary to find parallelism to effectively utilize the transistors. Currently, processor designs dynamically extract parallelism with these transistors by executing many instructions within a single, sequential program in parallel.

However, reliance on a single thread of control limits the parallelism available for many applications and the cost of extracting parallelism from a single thread is becoming prohibitive[8].

The demand for ever faster computer systems seems to be insatiable. Instruction-level parallelism helps a little, but pipelining and superscalar operations rarely win more than a factor of five or ten. To get gains of 50, 100 or even more, the only way is to design computers with multiple CPUs. Thus high level of gain is only promised by parallelism at the processor level. Traditionally the processor level parallelism has used discrete processors. Making one processor master and run the OS is attractive as it solves most of the previously cited problems, but is prone to the bus latencies and hence poor performance.

Researchers have proposed two microarchitectures that exploit multiple threads of control: Simultaneous Multithreading (SMT) and Chip Multiprocessors (CMP). From a purely architectural point of view, the SMT processor's flexibility makes it superior. However, the need to limit the effects of interconnect delays, which are becoming much slower than transistor gate delays, will also drive the billion-transistor chip design. Interconnect delays will force the microarchitecture to be partitioned into small, localized processing elements. For this reason, the CMP is much more promising because it is already partitioned into individual processing cores[8].

Programmers must find thread level parallelism in order to maximize CMP performance. With current trends in parallelizing compilers, multithreaded operating systems and awareness of programmers about how to program parallel computers, this problem should prove less daunting in future. Additionally, having all of the CPUs on a single chip allows designers to exploit thread-level parallelism even when threads communicate frequently.

SPE architecture: In designing our CMP we have used a modified form of Chip Multiprocessors (CMP). The new microprocessor architecture consists of two tightly coupled microprocessors. Both are able to communicate with each other directly and are implemented as a single unit on a single FPGA.

One of the microprocessors is the master processor and implements privileged instruction as well as rest of the instruction set. Operating system alone runs on this microprocessor. The second microprocessor only implements the non-privileged instructions. Complex processor execution units like floating point units and vector units are shared among both processors to avoid complex design and wastage of physical resources.

Only a single slave processor and no complex execution units as well as no caches were implemented to simplify the research effort.

MC-CPU instruction set architecture: MC-CPU instruction set was deigned from ground up to accommodate the new features of this architecture. It is a 32 bit RISC ISA. The SPE implements the MC-CPU Instruction Set Architecture. SPE consists of Master CPU (CPU-1), Slave CPU (CPU-2) and Shared Stack as shown in the Fig. 1. Each of these functional units and their operations are explained individually in the following subsections.

CPU 1: CPU 1 is the master CPU. It implements all the privilege instructions. Only the master processor can access I/O devices. Interrupt handling is performed only by the master processor. Shared stack is also controlled by master processor. The master processor can control the behavior of the slave processor by the means of INTS interrupts. Slave processor implements special interrupt handlers for INTS rather than for the normal interrupts.

CPU 2: CPU 2 is the slave CPU. It does not implement the privilege instructions. The slave processor can not physically access I/O devices. System level interrupt handling is not performed by the Slave processor as it does not have an INT line. Shared stack is accessed by the slave processor when the master processor grants it access. Slave processor implements special interrupt handlers for INTS rather than for the normal interrupts. When the master processor asserts INTS, the slave processor immediately jumps to the particular interrupt based on INTSCODE.

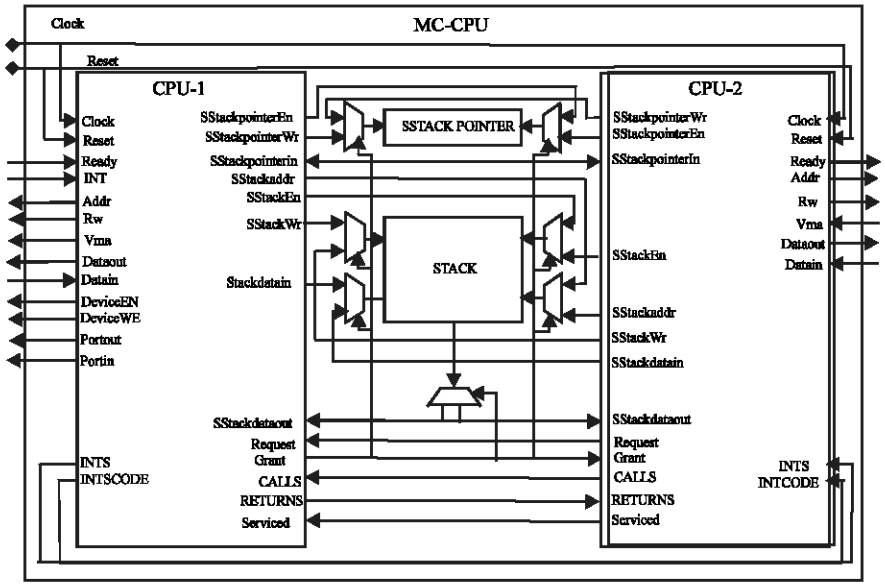

Fig. 1: SPE processor

Shared stack: Shared Stack consists of the stack memory and the shared stack pointer. The shared stack pointer is a 32 bit register. Its output value is constantly supplied to both processors. It can only be modified by one processor at any given time. Shared stack memory consists of single port 1024 bit RAM, arranged as 32 X 32 bits. Only one processor can push or pop from the shared stack at any given time. Shared stack operates in the following manner:

- Master processor has initial control of the shared stack
- Master processor can modify the shared stack pointer any time; only exception is when it has granted control of the shared stack to the slave processor
- Master processor can push or pop values from the shared stack any time; except when it has granted control of the shared stack to the slave processor
- Slave processor can not directly access the shared stack
- Slave processor must assert the REQUEST Signal to gain access to the shared stack
- Whenever REQUEST is asserted by the Slave processor, the master processor can grant or disallow access to the shared stack
- Access is disallowed only when master processor is modifying or accessing the shared stack itself
- Slave processor is blocked or in a wait state during this period
- When the master processor is not accessing the shared stack and the Slave processor requests for it, request is granted by asserting the GRANT signal
- When the GRANT signal is asserted, Slave processor gets access to the shared stack
- Slave processor can now modify both shared stack pointer and shared stack
- After the slave processor has modified the stack it deasserts the REQUEST signal to indicate that the shared stack is now free
- When the GRANT signal is deasserted the master processor deasserts the GRANT signal and takes the control of shared stack back

Remote call: All the communication between the master and slave processor is based on remote calls. In fact these are not remote calls in the classic sense rather these are traps to the OS kernel running on the master processor. Only the slave processor can trap to the master processor by asserting CALLS signal. The remote calls work in the following manner.

- When the user code running on the Slave processor needs some operating system service it must invoke a remote call
- Remote call is invoked by the slave processor by asserting the CALLS signal
- Before asserting the CALLS signal slave processor must request access to the shared stack and at least place the 32 bit service code on top of the shared stack. It can also place any parameters on the stack if there is any
- After placing the service code and/or any parameters on the shared stack, the Slave processor asserts the CALLS signal

- On receiving the CALLS signal Master processor invokes the remote call handler
- Remote call handler checks for user access rights and proper parameters and then calls the appropriate OS function. This is a normal function call
- Normally no context switch takes place during this whole procedure
- After servicing the call and placing return values onto the shared stack the Master processor asserts the RETURNS signal
- On receiving the RETURNS signal, Slave processor request for the shared stack, gets the return values and deasserts the CALLS signal

Remote interrupt: The Master processor controls the slave processor by using Remote Interrupts. Only the Master processor can raise remote interrupts and only the slave processor serves remote interrupts. Interrupt factor table and interrupt service routines for the remote interrupts are placed in the Slave processor's memory space by the Master processor. These interrupts can range from memory management to context switching to process cleanup. Remote interrupts work in the following manner.

- Operating System running on the Master processor can raise remote interrupts
- A remote interrupt is raised by asserting the INTS signal
- Interrupt type is indicated by INTSCODE
- Upon receiving an INTS the Slave processor immediately jumps to the appropriate handler based on INTSCODE
- After servicing the INTS the slave processor assert the SERVICED signal
- Upon receiving the SERVICED signal the Operating System on the Master processor considers the work done and deasserts the INTS signal

Experimental setup: In order to test the SPE processor it was necessary to have a complete computer system with all microprocessor support devices designed and implemented. So, memory, Input and Output multiplexers, a VGA controller, a Keyboard controller and an interrupt controller were also designed and implemented along with the SPE processor.

The whole system (Fig. 2) is implemented as a SoC on a single FPGA. The complete system utilizes approximately 95% of a Spartan-IIE.

The SPE processor achieved a clock speed of 25 MHZ. The system was mounted on a system board based on the arrangement shown in Fig. 3. It was interfaced with the computer using the parallel port. A test program was used to test proper operation of the system.

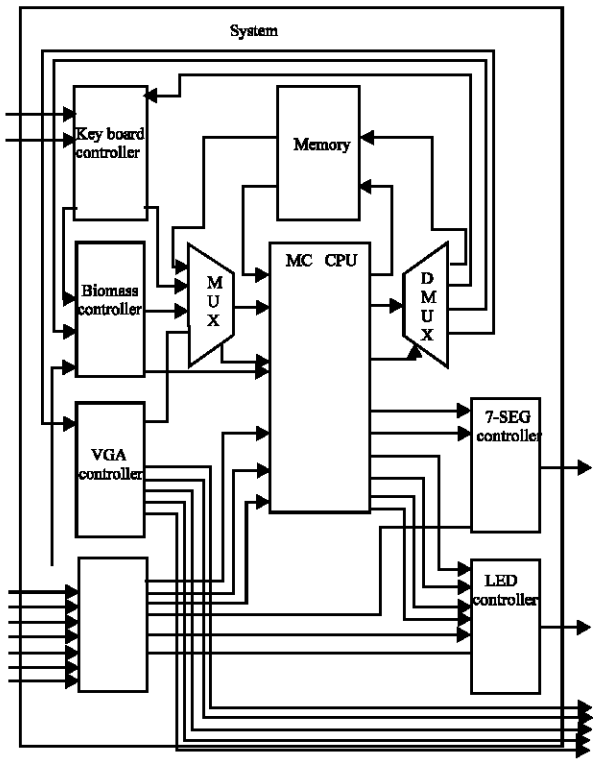

Fig. 2: System architecture diagram

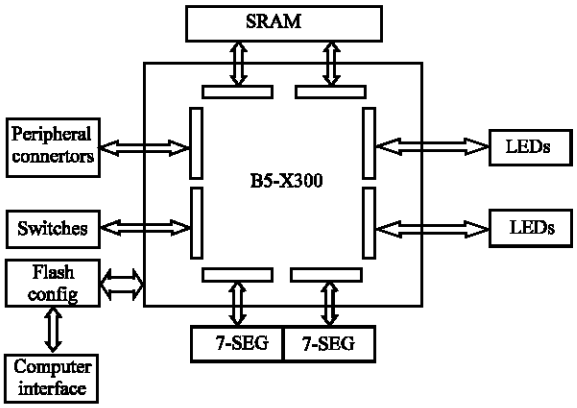

Fig. 3: Hardware arrangement

RESULTS AND DISCUSSION

As a result of this research project we have been able to verify the benefits of the MultiCore design. Specifically, a marked reduction in the context switch penalty. Since, the code running on the master processor is never preempted; it is able to service user requests more efficiently and quickly.

When compared to the SMP systems the outcome is very clear, the main bottle neck is the Interprocessor communication buss. In case of SPE there is no such

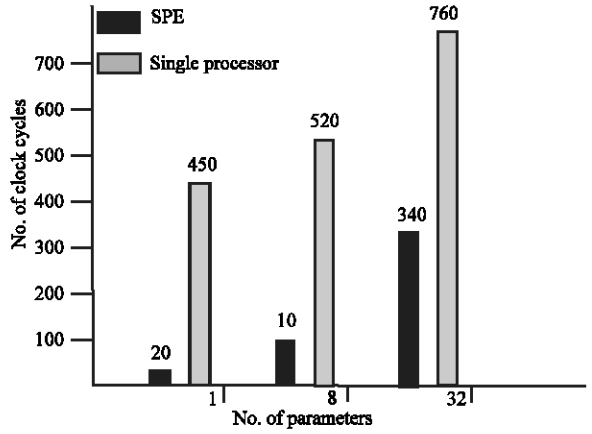

Fig. 4: No. of parameters versus clock cycles in a context switch

external Interprocessor communication buss and hence such latencies are avoided altogether.

When compared to the other CMP processors the SPE architecture does not employ any packet protocol for communication among the two processors. This improves the inter processor communication capability significantly. The downside is that it requires extensive hardware support.

The direct measure of SPE processor performance comes from comparing a piece of code that calls OS services, first on the Master CPU and then on the slave CPU.

When the code is run on the master processor, the system behaves just like a normal single processor system. At every system call performed by the user routine there is a context switch and the OS is switch back. The OS performance the necessary operation and then preempts itself while making the user program active.

In a context switch 43 registers are saved to memory. Saving a single register to memory takes 10 clock cycles. Saving 43 registers requires 430 clocks. In all a single context switch takes about 450 clock cycles on the master processor. This is for an OS service that only requires the service code and no parameters.

On the SPE processor, when the user code is run on the slave processor and it performs a system call then there is no need for a context switch to occur since the OS is running on a separate processor. A remote trap that only passes the service code to the master processor requires only 20 cycles.

It can be easily seen from Fig. 4 that a single context switch requires at least 500 clock cycles whereas a remote trap only requires 10 clock cycles. Thus it can be safely concluded that incorporating features at the microarchitecture level can improve IPC performances significantly. Improvements in IPC performance improve OS performance significantly.

FUTURE WORK

As next to investigating possible extensions, we are currently developing compiler tools capable of handling the offered flexibility. The ultimate goal would be to develop tools that enable fast software compilation by mapping specific performance requirements of an application into partitioned code. On of the code partitions will run on the master processor as a service for the bulk of code running on the slave processors.

Our future research will also concentrate on a method for analyzing which configuration of master processor and Slave processors will meet the requirements for a specific application in an optimal way.

REFERENCES

1. Brown, A.B., 1997. A Decompositional Approach to Computer System Performance Evaluation. Center for Research in Computing Technology Harvard University Cambridge, Massachusetts.
2. Intel Corp., 1993. Microprocessors Volume II, pp: 7-1-7-30 and 7-90-7-111.
3. Ratham, S. and G. Slavenburg, 1996. An architectual overview of the programmable multimedia processor TM-1. In: Proceedings of COMPCON 96, Santa Clara, California, USA, 25– 28 Febraury, IEEE Computer Society, Los Alamitos, California, USA., pp: 319-326.
4. Theelen, B.D. and A.C. Verschuere, 2003. Architecture design of a scalable single-chip multi-processor. J. Sys. Architecture, Special Issue on Systems and Verification, 49: 619-639.
5. Theelen, B.D., A.C. Verschueren, V.V. Suarez, M.P.J. Stevens andA. Nunez, 2003. A scalable single-chip multi-processor architecture with on-chip RTOS kernel. J. Sys. Architecture, pp: 619-639.
6. Bergamaschi, R., I. Bolsens, R. Gupta, R. Harr and A. Jerraya *et al*., 2001. Are Single-chip Multiprocessors in Reach? In: Wolf W, K. Roy Eds.), IEEE Design and Test of Computers, IEEE Computer Society, Los Alamitos, California, USA, 18: 82-89.
7. Liedtke, J., U. Dannowski, K. Elphinstone, G. Liefländer and E. Skoglund *et al*., 2001. The L4Ka Vision (White paper).
8. Lance, H., B.A. Nayfeh and K. Olukotun, 1997. A Single-Chip Multiprocessor. In: Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, pp: 2-11.