IFERP
connecting engineers... developing research

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)
Vol 3, Issue 12, December 2016

Simultaneous Multithreading for Superscalars

[1] Aishwarya P. Mhaisekar [2] Shubham S. Bokilwar [3] Anup Pachghare
Jawaharlal Darda Institute of Engineering and Technology, Yavatmal, 445001

Abstract: — **As the processor community is trying to approach high performance gain from memory. One approach is to add more memory either cache or primary memory to the chip. Another approach is to increase the level of systems integration. Although integration lowers system costs and communication latency,the overall performance gain to application is again marginal.In general the only way to significantly improve performance is to enhance the processors computational capabilities, this means to increase parallelism. At present only certain forms of parallelisms are being exploited .for ex .can execute four or more instructions per cycle; but in practice, they achieved only one or two because, in addition to the low instruction level parallelism performance suffers when there is little thread level parallelism so, the better solution is to design a processor that can exploit all types of parallelism. Simultaneous multi-threading is processor design that meets these goals. Because it consumes both threads level & instruction level parallelism.**

Key words:**--Parallelism, Simultaneous Multithreading, thread, context switching, Superscalar**

I. INTRODUCTION

In processor world, the execution of a thread is a smallest sequence of programmed instructions that can be managed by scheduler independently, which is a part of a operating system. In most of the cases a thread is a part or a component of a process. But if we see the implementation of thread and processes, they both differ between the operating systems. One process can contain multiple threads; executing parallel and it share the resources such as memory. But this doesn't happen in case of process.

The multithreading concept generally being implemented in a systems having single processor by time slicing. The CPU (Central Processing Unit) switches between different software threads. This switching is called as context switching. Due to this the users perceive the threads and threads and tasks as running in parallel.

Multithreading is mainly found in multitasking operating systems. Multithreading is a widespread programming and execution model that allows multiple threads to exist within the context of one process. These threads share the process's resources, but are able to execute independently. The threaded programming model provides developers with a useful abstraction of concurrent execution. Multithreading can also be applied to one process to enable parallel execution one multiprocessingsystem. Multithreaded applications have the following advantages, Responsiveness, Faster execution , Lower resource consumption, Better system utilization, Simplified sharing and communication, Parallelization.

Threads in the same process share the same address space. This allows concurrently running code to couple tightly and conveniently exchange data without the overhead or complexity of an IPC. When shared between threads, however, even simple data structures become prone to race conditions if they require more than one CPU instruction to update: two threads may end up attempting to update the data structure at the same time and find it unexpectedly changing underfoot. To prevent this, One concept reviewed in this paper that is Simultaneous multithreading.

Simultaneous multithreading (SMT) is a technique for improving the overall efficiency of superscalar CPUs with hardware multithreading. SMT permits multiple independent threads of executionto better utilize the resources provided by modernprocessor architectures. The name multithreading indicates multiple meanings, because not only can multiple threads be executed simultaneously on one CPU core, but also multiple tasks (with different Page tables, different Task state segments, different Protection rings, different IO permissions, ...) are also running on the same core, they are completely separated from each other.

The technique is really an efficiency solution and there is inevitable increased conflict on shared resources, measuring or agreeing on the effectiveness of the solution can be difficult.

IFERP
connecting engineers... developing research

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)
Vol 3, Issue 12, December 2016

II. THREAD VS. PROCESS

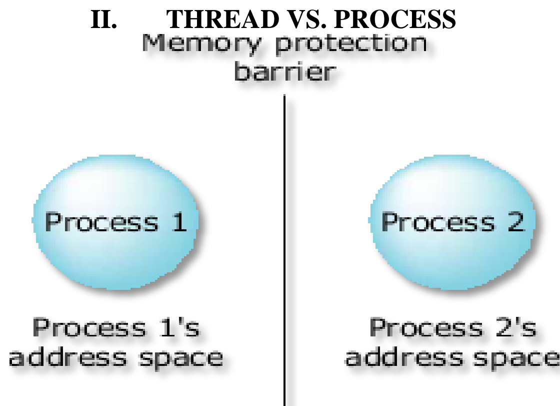

Fig 1. Memory Protection Barrier in Process

- Processes are typically independent, while threads exist as a component ofa process[9].
- Processes carry considerably more state information than threads, whereas multiple threads within a process share process state as well as memory and other resources[9].
- Processes have separate address spaces, whereasthreads share their address space[9].
- Processes interact only through system-provide inter-process Communication mechanisms[9].
- Context switching between threads in the same process is typically faster than context switching between processes[9].

III. SINGLE THREADING

In computer programming, single threading is nothing but the processing of single command at a time. The opposite of single threading is multithreading. While it has beensuggested that the term single threading is misleading, the term has been widely used within the functional programming community.

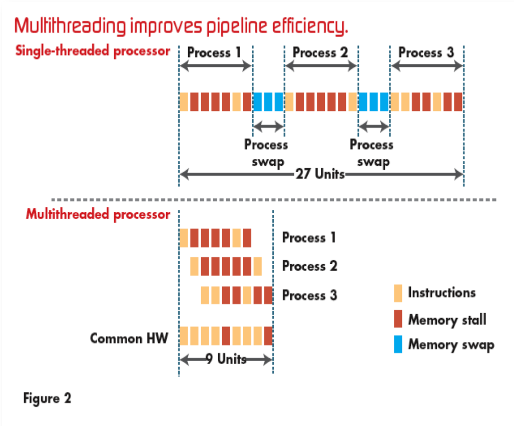

Fig 2.Working of Single and Multithreaded Processor

IV.MULTITHREADING

The minimal requirement for a multithreaded processor is the ability to get two or more threads of control in parallel within the processor pipeline –i.e. it must provide two or more independent program counters –and a mechanism that triggers a thread switch. Thread switch overhead must be very low, from zero to only a few cycles. A fast context switch is supported by multiple program counters and often by multiple register sets on the processor chip.
The principle approaches to multithreaded processor exits.

4.1. Interleaved Multithreading Technique –

An instruction of another thread is fetched and fed in to the execution pipeline at each processor cycle [1]

4.2.Block Multitheading Technique-

This instruction of a thread are executed successively until an event occurs that may cause latency .this event induces a context switch[3]

4.3.Simultaneousmultithreading-

The wide superscalar instruction issue is combined with the multiple –context approach. Instructions are simultaneously issued from multiple threads to the execution units +960 of a superscalar processor[8].

IFERP connecting engineers... developing research

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)
Vol 3, Issue 12, December 2016

V. PROCESSES, KERNEL THREADS, USER THREADS, AND FIBERS

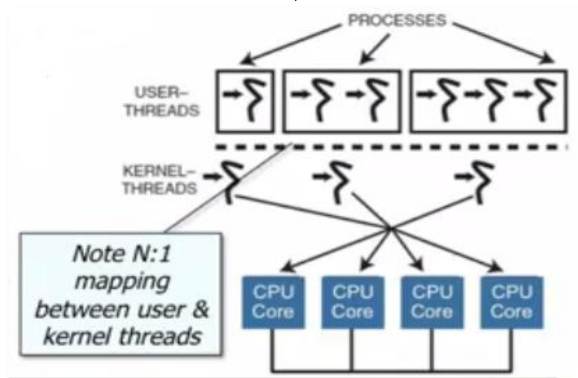

Fig 3. Mapping between User and Kernel Thread

Scheduling can be done at a kernel level or user level, and multitasking can be done preemptively or cooperatively. At the kernel level, a process contains one or more kernel threads, which share the process's resources, such as memory and file handles – a process is a unit of resources, while a thread is a unit of scheduling and execution. Kernel scheduling is typically uniformly done preemptively or, less commonly, cooperatively[11]. At the user level a process such as a runtime systemcan itself schedule multiple threads of execution. If these do not share data, as in Erlang, they are usually analogously called processes, while if they share data they are usually called (user) threads, particularly if preemptively scheduled. Cooperatively scheduled user threads are known as fibers; different processes may schedule user threads differently. User threads may be executed by kernel threads in various ways (one-to-one, many-to-one, many-to-many).

A process is a "heavyweight" unit of kernel scheduling, as creating, destroying, and switching processes is relatively expensive. Processes own resources allocated by the operating system. Resources include memory (for both code and data), file handles, sockets, device handles, windows, and a process control block. Processes are isolated by process isolation, and do not share address spaces or file resources. A kernel thread is a "lightweight" unit of kernel scheduling. At least one kernel thread exists within each process. If multiple kernel threads can exist within a process, then they share the same memory and file resources. Kernel threads are preemptively multitasked if the operating system's process scheduler is preemptive. Kernel threads do not own resources except for a stack. The kernel can assign one thread to each logical core in and can swap out threads that get blocked. However, kernel threads take much longer than user threads to be swapped[10].

Threads are sometimes implemented in user space libraries, thus called user threads. The kernel is not aware of them, so they are managed and scheduled in user space.

Fibers are an even lighter unit of scheduling which are cooperatively scheduled: a running fiber must explicitly beneficial to allow another fiber to run, which makes their implementation much easier than kernel or user threads. A fiber can be scheduled to run in any thread in the same process. This permits applications to gain performance improvements by managing scheduling themselves, instead of relying on the kernel scheduler[10].

VI.THREAD AND FIBER ISSUES

6.1. Concurrency And Data Structures

Threads in the same process share the same address space. This allows concurrently running code to couple tightly and conveniently exchange data without the overhead or complexity of an IPC. When shared between threads, however, even simple data structures become prone to race conditions if they require more than one CPU instruction to update: two threads may end up attempting to update the data structure at the same time and find it unexpectedly changing underfoot. Bugs caused by race conditions can be very difficult to reproduce and isolate[11].

To prevent this, threading application programming interfaces(APIs)offer synchronization primitives such as mutexes to lockdata structures against concurrent access. On uniprocessor systems, a thread running into a locked mutex must sleep and hence trigger a context switch. On multi-processor systems, the

thread may instead poll the mutex in a spinlock. Both of these may sap performance and force processors in symmetric multiprocessing (SMP) systems to contend for the memory bus, especially if the granularity of the locking is fine [9].

Although threads seem to be a small step from sequential computation, in fact, they represent a huge step. They discard the most essential and appealing properties of sequential computation: understandability, predictability, and determinism. Threads, as a model of computation, are wildly non-deterministic, and the job of the programmer becomes one of pruning that nondeterminism.

6.2 Synchronization

Since threads share the same address space and the same resources, the programmer must be very careful to avoid race conditions and other non-acceptable behaviors. In order for data to be correctly manipulated, threads will often need to rendezvous in time in order to process the data in the correct order. Threads may also require mutually exclusive operations (often implemented using semaphores) in order to prevent common data from being simultaneously modified or read while in the process of modification. Careless use of such primitives can lead to deadlocks [7].

6.3 Scheduling

For avoiding the problems like deadlocks one concept called as Scheduling is used. Operating systems schedule threads either preemptively or cooperatively. Preemptive multithreading is generally considered the superior approach, as it allows the operating system to determine when a context switch should occur. Cooperative multithreading, on the other hand, relies on the threads themselves to release control once they are at a stopping point. This can create problems if a thread is waiting for a resource to become available[10].

Until the early 2000s, most desktop computers had only one single-core CPU, with no support for hardware threads, although threads were still used on such computers because switching between threads was generally still quicker than full-process context switches. In 2002, Intel added support for simultaneous multithreading to the Pentium 4 processor, under the name hyper-threading; in 2005, they introduced the dual-core Pentium D processor and AMD introduced the dual-coreAthlon 64 X2 processor.

VII. SIMULTANEOUS MULTITHREADING

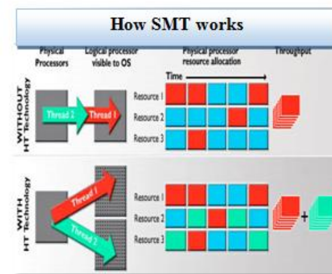

Fig 4. How SMT works

Multithreading is similar in concept to preemptive multitasking but is implemented at the thread level of execution in modern superscalar processors[2].

Simultaneous multithreading (SMT) is one of the two main implementations of multithreading, the other form being temporal multithreading. In temporal multithreading, only one thread of instructions can execute in any given pipeline stage at a time. In simultaneous multithreading, instructions from more than one thread can be executed in any given pipeline stage at a time. This is done without great changes to the basic processor architecture: the main additions needed are the ability to fetch instructions from multiple threads in a cycle, and a larger register file to hold data from multiple threads[4]. The number of concurrent threads can be decided by the chip designers. Two concurrent threads per CPU core are common, but some processors support eight concurrent threads per core. Simultaneous multithreading is a processordesign that meets this goal, because it consumesboth thread-level and instruction-levelparallelism. In SMT processors, thread-levelparallelism can come from either multithreaded,parallel programs or individual,independent programs in a multiprogramming workload. Instruction-level parallelismcomes from each single program or thread.Because it successfully (and simultaneously)exploits both types of parallelism, SMT processors use resources more efficiently, and both instruction throughput and speedups aregreater.

IFERP
connecting engineers... developing research

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)
Vol 3, Issue 12, December 2016

Simultaneous multithreading adds minimal hardware complexity to, and, in fact, is a straightforward extension of, conventional dynamically scheduled superscalars. Hardware designers can focus on building a fast, single threaded superscalar, and add SMT's multithread capability on top.

Figure 5 (a) shows a sequence from a conventional superscalar. As in all superscalars, it is executing a single program, or thread, from which it attempts to find multiple instructions to issue each cycle. When it cannot, the issue slots go unused, and it incurs both horizontal and vertical waste[5].

Figure 5 (b) shows a sequence from a multithreaded architecture, such as the Tera. Multithreaded processors contain hardware state (a program counter and registers) for several threads.The primary advantage of multithreaded processors is that they better tolerate long latency operations, effectively eliminating vertical waste.

SMT can recover issue slots lost to both horizontal and vertical waste. We derived our SMT model from a high-performance, out of- order; superscalar architecture whose dynamic scheduling core is similar to that of the Mips R10000.Simultaneous multithreading needs no special hardware to schedule instructions from the different threads onto the functional units.However, should an SMT implementation negatively impact either the targeted processor cycle time or the time to design completion, designers could take several approaches to simplify it.

VIII. SMT vs. MULTIPROCESSORS

Threads	Multiprogramming workload			Parallel workload				
	SS	FGMT	SMT	SS	MP2	MP4	FGMT	SMT
1	2.7	2.6	3.1	3.3	2.4	1.5	3.3	3.3
2	—	3.3	3.5	—	4.3	2.6	4.1	4.7
4	—	3.6	5.7	—	—	4.2	4.2	5.6
8	—	2.8	6.2	—	—	—	3.5	6.1

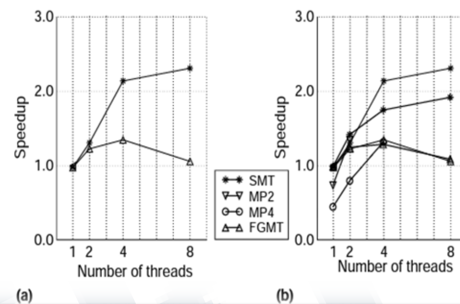

Fig 5. Multiprogramming and Parallel workload

SMT obtained better speedups than the multiprocessors (MP2 and MP4), not only when simulating the machines at their maximum thread capability (eight for SMT, four for MP4, and two for MP2), but also for a given number of threads. At maximum thread capability, SMT's throughput reached 6.1 instructions per cycle, compared with 4.3 for MP2 and 4.2 for MP4[6].

Speedups on the multiprocessors were hindered by the fixed partitioning of their hardware resources across processors, which prevents them from responding well to changes in instruction- and thread-level parallelism. Processors were idle when thread-level parallelism was insufficient; and the multiprocessor's narrower processors had trouble exploiting large amounts of instruction-level parallelism in the unrolled loops of individual threads. An SMT processor, on the other hand, dynamically partitions its resources among threads, and therefore can respond well to variations in both types of parallelism, exploiting them interchangeably. When only one thread is executing, (almost) all machine resources can be dedicated to it; and additional threads (more thread-level parallelism) can compensate for a lack of instruction-level parallelism in any single thread[2]. To understand how fixed partitioning can improve multiprocessor

performance, measured the number of cycles in which one processor needed an additional hardware resource and the resource was idle in another processor. (In SMT, the idle resource would have been used.) Fixed partitioningof the integer units, for both arithmetic and memory operations was responsible for most of MP2's and MP4's inefficient resource use. The floating-point units were also a bottleneck for MP4 on this largely floating-point-intensive workload. Selectively increasing the hardware resources of MP2 and MP4 to match those of SMT eliminated a particular bottleneck. However, it did not improve speedups, because the bottleneck simply shifted to a different resource. Only when we gave each processor within MP2 and MP4 allthe hardware resources of SMT did the multiprocessors obtain greater speedups. However, this occurred only when the architecture executed the same number of threads; at maximum thread capability, SMT still did better. The speedup results also affect the implementation of these machines. Because of their narrower issue width, the multiprocessors could very well be built with a shorter cycle time[6]. The speedups indicate that a multiprocessor's cycle time must be less than 70% of SMT's before its performance is comparable.

Future Application

This paper is reviewed to evaluate the applicability and efficiency of Simultaneous Multithreading (SMT) as the base architecture of a network processor. Indeed, the SMT model inherently allows the multiple parallel threads which must be dealt with in network processor applications. In this paper, we reviewed the architectural implications of network applications on the SMT architecture. We reviewed that, when executed as independent threads, applications chosen from different network layers show an improved Instructions Per Cycle (IPC) and cache behavior when compared with the situation where the program executed comes from a single network application. Finally, a new architectural solution to cope with packet dependency is reviewed.

REFERENCES

1. Alverson R. et al. (1990) The Tera Computer System. In proc. Int. Conf. Supercomputing, Amsterdam, The Netherland, June, pp. 1-6

2. K. Farkas et al., "The Multicluster Architecture: Reducing Cycle Time Through Partitioning," to appear in Proc. 30th Ann.IEEE/ACM Int'l Symp. Microarchitecture, IEEE Computer Society Press, Los Alamitos, Calif., Dec. 1997.

3. Kreuzinger, J. and Ungerer, T. (1990) Context-Switching techniques for decoupled multithreaded processors. In proc. 25th Euromicro Conf. Milano, Italy, September 4-7, pp 1:248-251. IEEE Computer Society Press, Los Alamitos, CA.

4. R. Alverson et al., "The Tera Computer System," Proc. Int'l Conf. Supercomputing, Assoc. of Computing Machinery, N.Y., 1990,pp. 1-6.

5. S. McFarling, "Combining Branch Predictors," Tech. Report TN- 36, Western Research Laboratory, Digital Equipment Corp., Palo Alto, Calif., June 1993.

6. S. Palacharla, N.P. Jouppi, and J.E. Smith, "Complexity-Effective Superscalar Processors," Proc. Int'l Symp. Computer.

7. T. Linholm and F. Yellin. The Java Vitual Machine Specification. Addison Wesley, second edition, 1999.

8. Tremblay, M. et al. (2000) The MAJCArchitecture, ACM, 1997, pp. 206-218. architecture: a synthesis of Parallelism and scalability.IEEE Micro, 20, 12-25.