Multithreading

Lin Gao

cs9244 report, 2006

Contents

1 Introduction **5**

2 Multithreading Technology **7**
2.1 Fine-grained multithreading (FGMT) 8
2.2 Coarse-grained multithreading (CGMT) 10
2.3 Simultaneous multithreading (SMT) 11
2.4 Case study of implementing SMT 14

3 Crosscutting Issues **17**
3.1 Cache Contention 17
3.2 SMT and CMP 18
3.3 Speculative multithreading 19

Chapter 1

Introduction

Multithreading is first introduced to exploit thread-level paprallelism within a processor. As has already been noted, memory delay has become an important problem for computer performance. A cache miss can result in stalls range between tens of cycles to hundreds of cycles. Certain architectures increase the delay further due to the overhead of the coherency protocol and memory consistency model. The long latency introduced by a missed memory access can be hidden by another independent thread (may or may not belong to same process). Although the execution of individual threads may be degraded, the overall throughput has been improved.

The power of multithreading is not limited to speed up programs by overlapping threads execution. Another important use of multithreading is to promote utilization of existing hardware resources. The basic idea of multithreading is to allow multiple threads to share the functional units of a single processor in an overlapped fashion. Using the same example presented above, stalls related to a cache miss means not only waste of the cpu cycles (waiting for the date to be fetched), but also the pipeline bubbles and idle functional units. Multithreading allows another ready thread to fill in and avoids these idle resources, so as improves utilization of resources.

There is a clear trend in computer industry that more and more chips will be multi-core microprocessors. This trend also urges multithreading technology to get the full benefits of multiprocessors.

With multiprocessors, a system can be built that can actually execute more than one process at the same time. But problems stated above still exist, the number of pipeline bubbles is doubled when the number of processes that can simultaneously execute doubled, and the number of waste cpu cycles is doubled as well, if the stalls occur. So while multiprocessors

can improve performance by throwing transistors at the problem of execution time, the overall lack of increase in the execution efficiency of the whole system means that multiprocessor can be quite wasteful.

Multithreading become so important that hardware vendors have already implemented it in their commercial architectures, Pentium 4 Xeon is a good example. More details of multithreading will come in Chapter 2, and Chapter 3 will discuss some issues extended from multithreading.

Chapter 2

Multithreading Technology

Multithreading enables the thread-level parallelism (TLP) by duplicating the architectural state on each processor, while sharing only one set of processor execution resources. When scheduling threads, the operating system treats those distinct architectural states as separate "logical" processors. Logical processors are the hardware support for sharing the functional units of a single processor among different threads. There are several different sharing mechanism for different structures. The kind of state a structure stores decides what sharing mechanism the structure needs. Table 2.1 enumerates which resource falls into which category:

Category	Resources
Replicated	Program counter(PC) Architectural registers Register renaming logic
Partitioned	Re-order buffers Load/Store buffers Various queues, like the the scheduling queue, etc
Shared	Caches Physical registers Execution units

Table 2.1: Sharing Mechanisms

Replicated resources are the kind of resources that you just cannot get around replicating if you want to maintain two fully independent contexts on each logical processor. The most obvious of these is the program counter

(PC), which is the pointer that helps the processor keep track of its place in the instruction stream by pointing to the next instruction to be fetched. We need separate PC for each thread to keep track of its instruction stream.

Partitioned resources are mostly to be found in form of queues that decouple the major stages of pipeline from one another. There are two different kinds of partitioned resources, *statically partitioned resource* and *dynamically partitioned resource*. We will explain them in detail in section 2.4. No matter which category a partitioned resource belongs to, there are always some constrains for threads to obey, specifically the size of the resource any thread allowed to use.

Shared resources are at the heart of multithreading and makes the technique worthwhile. The more resources that can be shared between logical processors, the more efficient multithreading can be.

All multithreading approaches use similar resources partitioning system. Although there are similarities in the multithreading implementations, there are still some differences. Two major differences are 1) thread scheduling policy, and 2) pipeline partitioning. The first difference comes along with the question: when to switch from one thread to another? The answer to this question depends on what kind of latencies (specifically length of latencies) are going to be tolerated. Associated with the second difference is how exactly threads share the pipeline. Similarly, it related to how much single thread performance is willing to sacrifice.

Based on above criteria, there are three main approaches to multithreading: 1) Fine-grained multithreading (FGMT), 2) Coarse-grained multithreading (CGMT), and 3) Simultaneous multithreading (SMT). Figure 2.1 shows the differences among three approaches. Different color represents different thread and the white box means idle execution unit. At every cycle, four execution units can work in parallel. As illustrated in Figure 2.1 (a) for a single-thread processor, a lot of execution units are wasted because of lack of ILP. Three multithreading approaches use TLP to improve the resources utilization and hide the latencies caused by any event, the following section will discuss these three approaches in detail. And the last section in this chapter will investigate the implementation of SMT in a commercial processor.

2.1 Fine-grained multithreading (FGMT)

Fine-grained multithreading switches between threads on a fixed fine-grained schedule, usually processing instructions from a different thread on every

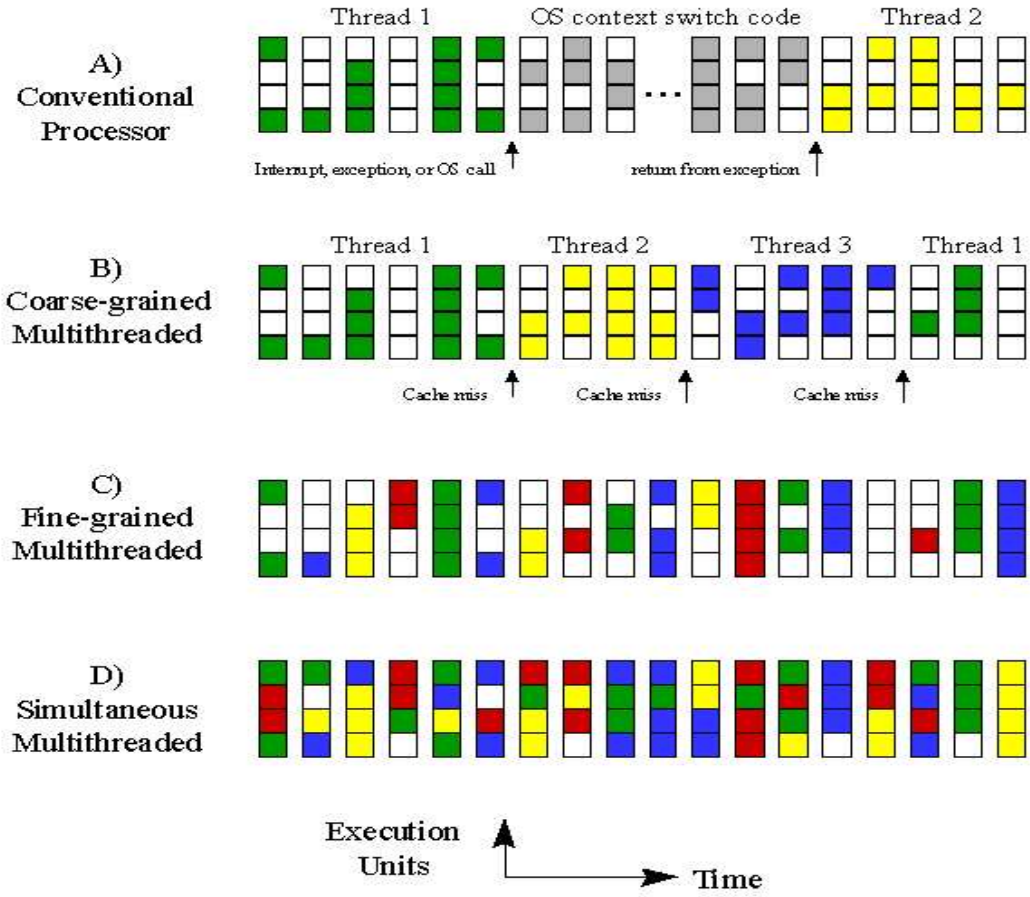

Figure 2.1: Three different approaches use the issue slots of a superscalar processor

cycle, causing the execution of multiple threads to be interleaved. Figure 2.1 (c) illustrates conceptually how FGMT works.

As illustrated above, this cycle-by-cycle interleaving is often done in a round-robin fashion, skipping any threads that are stalled due to branch mispredict or cache miss or any other reason. But the thread scheduling policy is not limited to the cycle-by-cycle (round-robin) model, other scheduling policy can also be applied too.

Although FGMT can hide performance loses due to stalls caused by any reason, there are two main drawbacks for FGMT approach:

- FGMT sacrifices the performance of the individual threads.
- It needs a lot of threads to hide the stalls, which also means a lot of register files.

There were several systems using FGMT, including Denelcor HEP and Tera MTA (both built by Burton Smith). Since the drawbacks listed above, sole FGMT is not popular today.

2.2 Coarse-grained multithreading (CGMT)

Coarse-grained multithreading won't switch out the executing thread until it reaches a situation that triggers a switch. This situation occurs when the instruction execution reaches either a long-latency operation or an explicit additional switch operation. CGMT was invented as an alternative to FGMT, so it won't repeat the primary disadvantage of FGMT : severely limits on single-thread performance.

CGMT makes the most sense on an in-order processor that would normally stall the pipeline on a cache miss (using CGMT approach, rather than stall, the pipeline is filled with ready instructions from an alternative thread).

Since instructions following the missing instructions may already be in the pipeline, they need to be drained from the pipeline. And similarly, instructions from new thread will not reach the execution stage until have traversed earlier pipeline stages. The cost for draining out and filling in the pipeline is considered as *thread-switch penalty*, and it depends on the length of the pipeline. So normally CGMT need short in-order pipeline for good performance.

Also because of the thread-switch penalty, CGMT is useful only for reducing the penalty of high cost stalls, where thread-switch penalty is negligible compared to the stall time. It then turns out to be CGMT's greatest

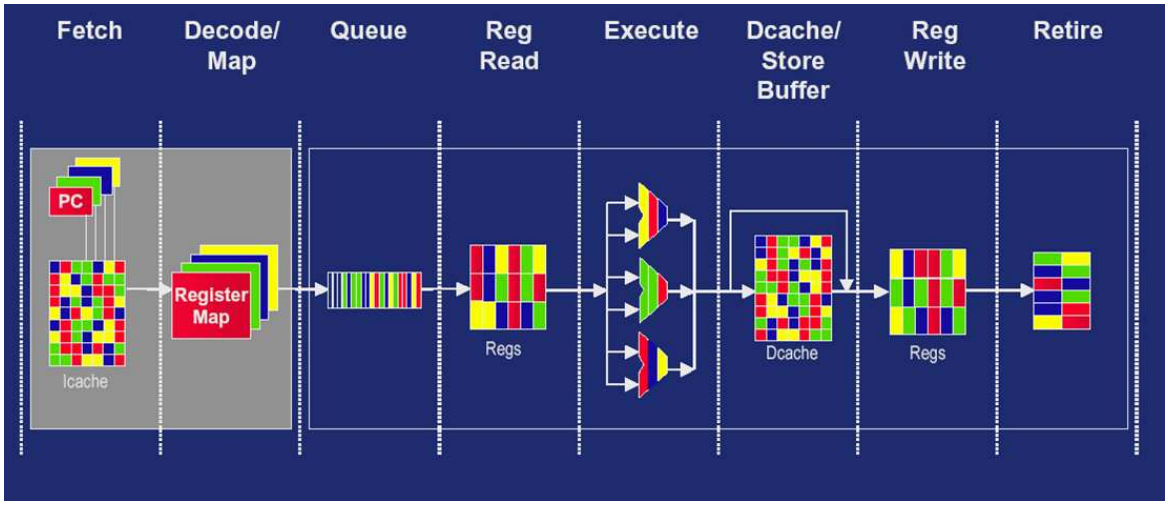

Figure 2.2: An example pipeline supporting SMT

weakness that it cannot tolerate short stalls. This limits its ability to improve the throughput and CGMT does suffer from the performance degradation when short latency instructions occur frequently.

Figure 2.1 (b) shows how CGMT exploits the resources of a superscalar. CGMT is commercialized in the Northstar and Pulsar Power PC processors from IBM.

2.3 Simultaneous multithreading (SMT)

Simultaneous multithreading is a fine-grained multithreading with dynamically varying interleaving of instructions from multiple threads across shared execution resources. The motivation behind the SMT is that, although FGMT and CGMT improve resources utilization by overlapping the latency of an instruction from one thread by the execution of another thread, there are still lots of white boxes (which are idle issue slots) in Figure 2.1 (b) and (c).

SMT share functional units dynamically and flexibly between multiple threads. Figure 2.2 gives an example pipeline supporting SMT. The pipeline is partitioned into two parts, the front-end (first two stages with gray shadow) and the back-end (pipeline stages that are thread-blind). Regardless of the difference between varied implementations of SMT, instruc-

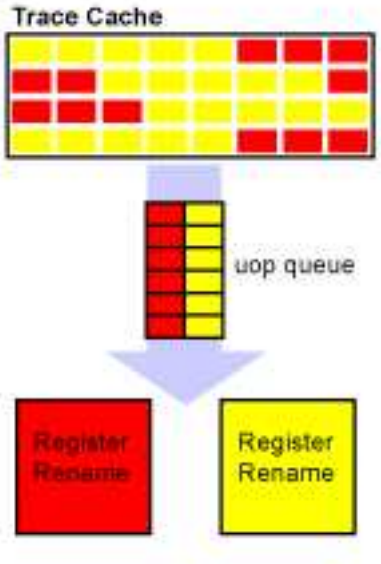

Figure 2.3: Statically Partitioned Queue

tions from multiple threads have to be joined before the pipeline stage where resources are shared. In our example, PC in $fetch$ stage and register map in *decode* stage are replicated. Architectural registers are renamed to share a common pool of physical registers, and the same architectural register from different threads are mapped to different physical registers, so instructions from multiple threads can be mixed in the data path without confusing sources and destinations across the threads. This leads to the insight that multithreading can be built on top of an out-of-order processor.

Pipeline stages from back-end speak a different story. These stages are thread-blind because they do not require the knowledge of SMT. For example, in the *execute* stage, the execution units don't know the difference between one thread from another when they execute instructions. An instruction is just an instruction to the execution units, regardless of which thread/logical processor it belongs to.

As mentioned at the beginning of this chapter, other than replicating some resources, a group of resources need to be partitioned between threads and mechanisms of partitioning falls into the following two types.

First, *statically partitioned* resource is simply decomposed equally into

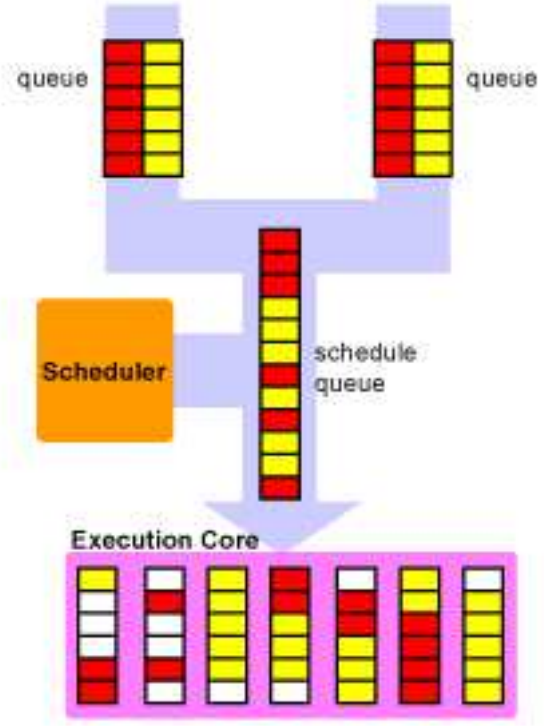

Figure 2.4: Dynamically Partitioned Queue

as many pieces as the number of logical processors it has. The uop queue in Pentium 4 Xeon [5] is a statically partitioned queue as showed in Figure 2.3. The uop queue is split in half, with half of its entries designated for the sole use of one logical processor and the other half designated for the sole use of the other.

Secondly, *dynamically partitioned* resource allows any logical processor to use any entry of that resource but it places a limit on the number of entries that any one logical processor can use. The scheduling queue in Xeon is a dynamically partitioned queue as showed in Figure 2.4. From the point of view of each logical processor and thread, this kind of dynamic partitioning has the same effect as fixed partitioning: it confines each logical processor to half of queue. However, form the point of view of the physical processor, there is a crucial difference. In this example, although scheduling queue is

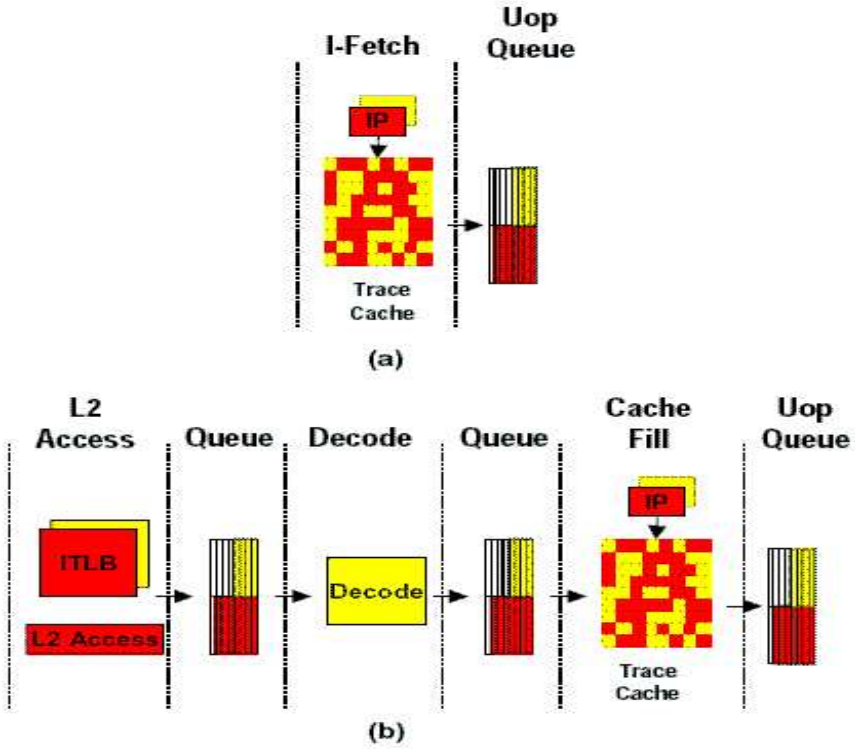

Figure 2.5: Xeon's front-end detailed pipeline

itself aware of the differences between instructions from one thread and the other, the scheduler, as a SMT-unaware shared resource, pull instructions out of the queue as if the entire queue holding a single instruction stream.

Like FGMT, SMT tolerates all latencies (both long stalls and short stalls), but it also sacrifices some individual threads' performance. SMT is implemented on a commercial processor Pentium 4 Xeon from Intel. We will further discuss the SMT implementation in next section.

2.4 Case study of implementing SMT

Simultaneous multithreading (SMT), also know as hyper-threading, might seem like a pretty large departure from the kind of conventional, process-switching multithreading done on a single-threaded CPU, it actually doesn't add too much complexity to the hardware. Intel reports that adding hyper-

threading to their Xeon processor [5] added only 5% to its die area. This section investigates the Xeon's SMT implementation.

Figure 2.5 along with Figure 2.6 give the full information of Xeon's detailed pipeline. Intel's Xeon is capable of executing at most two threads in parallel on two logical processors. Here are some details of major functions:

- Execution Trace Cache (TC) TC can only be accessed by one thread per cycle. If both logical processors want access to the TC at the same time, access is granted to one then the other in alternating clock cycles. The TC entries are tagged with thread information and are dynamically allocated as needed. The share nature of the TC allows one logical processor to monopolize it if only one logical processor is in running.

- ITLB and Branch Prediction The instruction translation lookaside buffer (ITLB) receives request from TC to deliver new instructions when a TC miss occurs, and it translates the next-instruction pointer (IP) address to a physical address. ITLB is replicated. The branch prediction structures are either replicated or shared: the return stack buffer and the branch history buffer is replicated because both structures are very small, and the large global history array is a shared structure with entries tagged with a logical processor ID.

- Instruction Decode Both logical processors share the decoder logic. If both threads are decoding instructions simultaneously, decoder logic alternates between threads in a coarse granularity. The switch policy is chosen in the interest of die size and to reduce complexity.

- Uop Queue Decoded instructions that written to TC and forwarded to uop queue, are called micro-operations or *uops*. This queue is statically partitioned in half as mention in Section 2.3. And this queue decouples the front-end from its out-of-order execution engine.

- Allocator The allocator logic assigns resources between uops, including the 126 re-order buffer (ROB) entries, 128 integer and 128 floating-point physical registers, 48 load and 24 store buffer entries. Some of the buffers are partitioned (ROB and load/store buffer).

- Register Rename The register rename logic renames the architectural IA-32 registers onto the machine's physical registers. This allows the 8 GPRs to dynamically share the 128 physical registers. It uses a register alias table (RAT) to track the mapping. RAT is replicated for each logical processor.

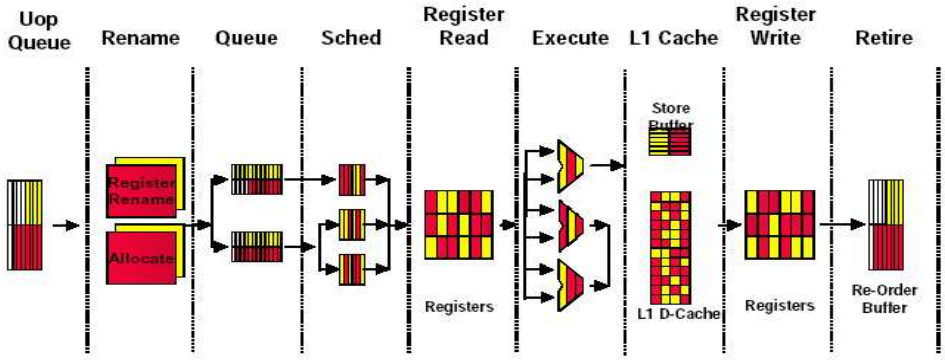

Figure 2.6: Xeon's out-of-order execution engine detailed pipeline

- Instruction Scheduling Five uop schedulers are used to schedule different types of uops for the various execution units. Each scheduler has its own scheduling queue of eight to twelve entries. The scheduler is SMT-unaware, while the scheduling queue is dynamically partitioned to avoid deadlock and ensure fairness as we've explained in Section 2.3.

- Retirement The retirement logic commits the architecture state in program order. If two uops from two logical processors respectively are ready to retired, the retirement logic retires the uops in program order for each logical processor by alternating between the two logical processores.

Xeon together with its hyperthreading technology allows software to run unmodified on two logical processors. While hyperthreading will not provide the level of performance scaling achieved by adding a second processor, benchmark tests show some server applications can experience a 30 percent gain in performance.

Chapter 3

Crosscutting Issues

3.1 Cache Contention

For a simultaneously multithreaded processor, the cache coherency problems associated with SMP all but disappear. Both logical processors on an SMT system share the same caches as well as the data in those caches. So if a thread from logical processor 0 wants to read some data that's cached by logical processor 1, it can grab that data directly from the cache without having to snoop another cache located some distance away in order to ensure that it has the most current copy.

However, since both logical processors share the same cache, the prospect of cache conflicts increase. This potential increase in cache conflicts has the potential to degrade performance seriously.

The fact that Xeon's two logical processors share a single cache does not implicates the cache size is effectively halved for each logical processor. Actually, each of the Xeon's caches–the trace cache, L1, L2, and L3–is SMT-unaware, and each treats all loads and stores the same regardless of which logical processor issued the request. So none of the caches know the difference between one logical processor and another, or between code from one thread or another. This means that one executing thread can monopolize virtually the entire cache if it wants to, and the cache, unlike the processor's scheduling queue, has no way of forcing that thread to cooperate intelligently with the other executing thread. The processor itself will continue trying to run both threads, by issuing fetched operations from each one. This means that, in a worst-case scenario where the two running threads have two completely different memory reference patterns (i.e. they're accessing two completely different areas of memory and sharing no data at all) the

cache will begin thrashing as data for each thread is alternately swapped in and out and bus and cache bandwidth are maxed out.

This kind of cache contention is observed that for some benchmarks SMT performs significantly worse than either SMP or non-SMT implementations within the same processor family, especially in the memory-intensive portion of the benchmark suite.

In summary, resource contention is definitely one of the major pitfalls of SMT, and it's the reason why only certain types of applications and certain mixes of applications truly benefit from the technique. With the wrong mix of code, hyper-threading decreases performance, just like it can increase performance with the right mix of code.

3.2 SMT and CMP

Chip Multiprocessors (CMPs) integrates multiple processor cores on a single chip, which eases the physical challenges of packing and interconnecting multiple processors. This kind of tight integration reduces off-chip signaling and results in reduced latencies for processor-to-processor communication and synchronization.

Both CMP and SMT [2, 7] are techniques for improving throughput of general-purposed processors by using multithreading. SMT is a technique that permits multiple independent threads to share a group of execution resources. It improves the utilization of processor's execution resources and provides latency tolerance. CMPs use relatively simple single-thread processor cores to exploit thread-level parallelism with one application by executing multiple threads in parallel across multiple processor cores.

From an architectural point of view, SMT is more flexible. However the inherent complexity of SMT architecture leads to the the following problems: 1) increasing the die area, 2) requiring longer cycle time. Since SMT architecture is composed of many closely interconnected components, including various buffers, queues and register files. I/O wires are required to interconnect these units and delay associated with these wires may delay the CPU's critical path.

On the other hand, CMP architecture uses relative simple single-thread processor (comparing to the complex hardware design of SMT), so it allows a fairly short cycle time. Also CMP is much less sensitive to poor data layout and poor communication management since the inter-core communication latencies are lower. From a software perspective, CMP is an ideal platform to run multiprogrammed workloads or multithreaded applications. However,

CMP architecture may lead to resource waste if an application cannot be effectively decomposed into threads or there is not enough TLP.

Since both SMT and CMP have their advantages and drawbacks, a lot researches have been done to improve both architectures, including a combined CMP/SMP architecture and also speculative multithreading. We will discuss speculative multithreading in the following section.

3.3 Speculative multithreading

Since a large number of applications show little thread-level parallelism, both hardware and software face the challenge to discover adequate TLP in this class of applications to get the benefits from multithreading architectures. Speculative multithreading (SpMT) [1, 4, 3, 6] is an attempt to explore TLP for sequential applications, especially for nonnumeric applications. SpMT releases threads execution from strict semantic order presented in the application, so thread running on one processor may be control or data dependent on another thread that is still in running on another processor.

The speculative parallel execution of threads in SpMT may lead to dependence violation, also recognized as mis-speculation. Hardware tracks logical order among threads to detect any dependence violation. When mis-speculation found, related threads need to rollback and be re-executed.

There are many challenges for the SpMT model, including:

- The proper mechanism to detect mis-speculation
- The proper mechanism to rollback (partially or fully?)
- How to efficiently identify threads
- How to calculate the benefits according to thread start-up overhead and cost of mis-speculation

Research shows that the potential performance gain for SpMT model is significant. Now it is the challenge for both hardware and software developers to turn the potential gain into a really improvement.

Bibliography

[1] Z. H. Du, C. Ch. Lim, X. F. Li, C. Yang, Q. Zhao, and T. F. Ngai. A cost-driven compilation framework for speculative parallelization of sequential programs. In *Proceedings of Conference on Programming Language Design and Implementation*, 2004.

[2] R. A. Dua and B. Lokhande. A comparative study of smt and cmp multiprocessors. *Project Report*, ee8365.

[3] V. Krishnan and J. Torrellas. Hardware and software support for speuculative execution of sequential binaries on a chip-multiprocessor. In *Proceedings of the 12th International Conference on Supercomputing*, pages 85–92. ACM Press, 1998.

[4] P. Marcuello and A. Gonzalez. A quantitive assessment of thread-level speculation techniques. In *Proceedings of the 15th Int. Parallel and Distributed Processing Symposium*, 2000.

[5] D. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty, J. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. In *Intel Technology Journal*, 2002.

[6] T. Ohsawa, M. Takagi, S. Kawahara, and S. Matsushita. Pinot: Speculative multi-threading processor architecture exploiting parallelism over a wide range of granularities. In *Proceedings of the 38th International Symposium on Microarchitecture (MICRO'05)*, 2005.

[7] R. Sasanka, S. V. Adve, Y. Chen, and E. Debes. Comparing the energy efficiency of cmp and smt architectures for multimedia workloads. In *UIUC CS Technical Report UIUCDCS-R-2003-2325*, 2003.