Thread-Parallel Architectures

Lecture #3: Tuesday, 8 April 2003
Lecturer: Metha Jeeradit and Wajahat Qadeer
Scribe: John Kim and Rebecca Schultz

1 Analysis Multi-threaded Architectures for Parallel Computing[1]

1.1 Summary and Original Contributions

This paper provides a model to characterize the behavior of a multithreaded machine based on 4 parameters including the memory access latency, the number of threads that can be interleaved, the cost of context switching, and the length of each thread. The first 3 of these parameters are architecture dependent and are determined by the machines interconnection, the amount of processor state available, and the switching mechanism used. The length of the thread, however, depends on both the architecture and the application behavior.

Two basic regions of operations are identified by the authors: a linear region and a saturation region. In the linear region, the processors efficiency is linear with the number of contexts that you have. On the other hand, in the saturation region, the efficiency is independent of both the number of contexts and the latency of the memory cycle. In this saturation region, the ratio between the switching cost and the length of the thread determines the maximum efficiency. However, when taking cache interference effects into account, having a large number of threads can hurt your efficiency in saturation.

1.2 Critique

The main strength of this paper is a practical, deterministic model available for use as a first approximation when starting to design a multithreaded architecture. With this model, the architectural designer can easily estimate the performance of his/her design before doing any simulation of the architecture in software.

The main critique is the debatable assumptions that they made:

1. They assume sufficient parallelism is available all the time. This is application dependent and this scenario seems unrealistic.

2. They also ignore synchronization issues which may substantially reduce your processor's efficiency.

3. They also assume a constant value for memory access latency which may not be true since you can have local cache, remote cache, local memory and remote memory, all of which will have different latencies.

1.3 Future Work

A natural future work to this paper is to extend the model by including the debatable assumptions that they have made.

2 Interleaving: A Multi-threading Technique Targeting Microprocessors and Workstations[2]

2.1 Summary and Original Contributions

This paper proposes a new hardware technique, called interleaving, to implement efficient multithreading. This technique suggests architectural improvements in commodity microprocessors for the efficient implementation of multithreading in both the workstation and multiprocessor loads. It is an extension of the fine-grained multithreading technique with the addition of data caches and pipeline interlocks for the implementation of multiple contexts while matching the performance of a single context microprocessor. Considerable improvement over blocked scheme has been shown for the interleaved scheme in both the workstation and multiprocessor environments due to its low switching cost and the ability to hide small latencies.

2.2 Critique

The main strength of this paper is the convincing results to substantiate the claim that interleaved multithreading architecture yields better performance than the blocked and fine grained schemes in both workstations and multiprocessors.

The main critiques include the significant hardware complexity for caches and program control unit for RISC based machines and the fact that interleaving technique is difficult to implement in dynamically scheduled superscalar processor.

2.3 Future Work

A possible future work from this paper is to research an efficient way to implement multithreading in the dynamic super scalar processor. Another possible future work includes finding a way to combine both interleaved and blocked approaches together to provide a low switching cost implementation while still providing the performance of a high priority thread comparable to a single-threaded architecture performance.

3 Comparative Evaluation of Latency Reducing and Tolerating Techniques[3]

3.1 Summary

This paper provides a consistent a consistent framework for the evaluation of the following techniques for multi-processor architectures:

- Coherent Caches
- Memory consistency models
- Software controlled prefetching
- Multiple contexts

The results of this paper show that using coherent caches offer substantial gains in the performance especially for avoiding read misses however, the hit rate is found to be lower than uni-processors. Relaxed memory consistency model is found to be better than sequential model due to its potential for improved performance however its complexity is higher. Pre-fetching and multiple contexts techniques are both very application dependent though they still provide some performance improvements when run separately.

3.2 Critique

The main strength of this paper is a systematic and consistent scheme to compare various latency tolerating schemes, providing a better understanding between their tradeoffs.

The main critiques of this paper include:

- In sufficient number of applications to substantiate the results presented in the paper.
- Results for the combined technique of prefetching and multiple contexts were not considered appropriately: The pre-fetching implementation (from the one used in single context) should have been modified before being used with the multiple context techniques.
- Lock-up free caches were also not exploited for read misses, which would have offered substantial performance gains for architectures supporting multiple-contexts.

3.3 Future Work

A natural future work to this paper is to compare these techniques against other latency tolerating schemes such as out-of-order execution and using DLP architecture.

4 Discussion

Multithreading is a technique for tolerating latency in the system. The various latency causing events include cache misses, synchronization penalties, TLB faults, and data/control dependencies.

Architectural latency avoiding or reducing techniques are caching, prefetching in hardware or software, out-of-order execution, exploiting data-level parallelism, relaxed memory coherence, and multithreading. Caching and prefetching works well for data sets which are regular and have locality, but for irregular memory accesses, multi context hardware method might be more beneficial.

In a multithreaded architecture, one important issue is to determine when to switch threads. The analysis paper[1] identifies the parameters L, the latency of a remote reference, and R the interval between switches triggered by a remote reference. When L $>$ C, it makes sense to switch threads.

A second issue is the whether it is possible to optimize software to minimize latency. The parameter L is hardware dependent, so software can only try to maximize R. This can be accomplished in part by spreading out long latency operations such as loads and stores. This can be done by using multiple contexts. These independent contexts are identified by the compiler in the approach in papers [1], [2], and [3], however, the requires a very sophisticated parallelizing compiler. In some cases threads can be generated even if independence can not be proven by the compiler, this approach is called speculative multithreading, and will be discussed further in lecture 4.

An additional question is how to reduce the cost of swapping registers on a context switch. Some methods for reducing this cost are having additional registers, having two sets of registers and saving and restoring one set in the background while using the other, using register renaming in hope that both contexts will fit in the register file, and maintaining dirty and valid state bits on the register file and swapping only those that are dirty.

In a deep pipeline, a thread may not block until late in the pipe. At this point the switching penalty would be very large. It would be useful to be able to predict the switch efficiently. One proposed solution is to use a hardware predictor similar in structure to a branch predictor to identify instructions that are likely to block. An alternative is to switch on every load or store, as these operations are potentially high latency, or to use the interleaving technique described in [2]. Interleaving, however, is not efficient when there is only a single thread and each thread may take a very long time to complete. An improvement would be to introduce prioritization with some OS assistance, however, care must be taken to avoid starvation.

Simultaneous-multithreading (SMT) is a different flavor of multithreading that extends traditional superscalar hardware to allow instructions to be issued from several threads every cycle to take advantage of idle functional units. However, SMT requires the hardware to check data dependencies, reorder commit, and do register renaming for each thread. As a result, the hardware cost of an SMT processor is very high, and very few applications have the necessary ILP to make it worthwhile. Essentially, the processor is turning thread level parallelism into instruction level parallelism, thus it often would be better to use a few cores rather than a single SMT processor. A SMT processor is only appropriate if a high level ILP is available and the hardware already exists, as is the case for a company like Intel that has already invested in designing a wide issue superscalar.

When thread level parallelism is exploited using multiple processors there is an additional issue of how to synchronize the concurrently executing threads. Synchronization in can be done in two ways : hardware or software. The software primitives that are generally used for synchronization are locks and barriers. Locks required that a thread "acquire" a particular lock before accessing the corresponding memory location. Only 1 thread is allowed to acquire the lock at a time. When a thread finishes using a memory location, it releases the lock. Barriers establish a point where a thread waits until all the threads reach that point, synchronizing the threads. Hardware implementations of synchronization are usually done with atomic instructions such as test and set, hardware barriers, disabling interrupts, and using full/empty bits. The full/empty bit approach allows for fine-grained producer-consumer synchronization. On a load instruction, if the bit is set to be full, the load is blocked. Similarly, if the bit is set to be empty on a store, the store is blocked.

A comparison was made between chip multiprocessor (CMP) and standard multiprocessor (MP) systems. CMP systems offer several benefits: lower latency, more bandwidth, and by using a snooping bus protocol, directory based synchronization may not be needed. In either case, it is necessary to identify regions of coherency. For CMP, it may be best to use incoherent L1 caches for private data, and store shared data in a coherent L2 cache. Contexts could be switched on a L2 miss. Additional savings can be made by identifying "read mostly" data that is read-write when the program first starts, but it switches to read-only during execution, and need not be synchronized.

Also, in CMP systems there is a question of homogeneous versus heterogeneous processors on the chip. Homogeneous systems are simpler to design software for, so homogeneous hardware is often programmed so that groups of cores have heterogeneous functions. For example, the same memory can be used as a cache for some applications or as a FIFO for others. Some functions are best accomplished using heterogeneous hardware, these can be identified and designed using a tool like the one provided by Tensilica.

In conclusion, a multicontext processor needs the following:

- Additional registers
- Non-blocking caches
- A hardware or software method for determining when to switch contexts

References

[1] (R) R. Saavedra-Barrera , D. Culler , T. von Eicken. "Analysis of Multithreaded Architectures for Parallel Computing". *Proceedings of the 2nd Symposium on Parallel Algorithms and Architectures (SPAA)*, Crete, Greece, July 1990.

[2] J. Laudon, A. Gupta, M. Horowitz. "Interleaving: A Multithreading Technique Targeting Multiprocessors and Workstations". *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Jose, CA, October 1994.

[3] A. Gupta, J. Hennessy , K. Gharachorloo , T. Mowry , W. Weber. "Comparative Evaluation of Latency Reducing and Tolerating Techniques". *Proceedings of the 18th International Symposium on Computer Architecture (ISCA)*, Toronto, Canada, May 1991