Speculative Multithreading

Lecture #4: Thursday, April 10 2003
Lecturer: Amin Firoozshahian and Brad Schumitsch
Scribe: Suzanne Rivoire and Ernesto Staroswiecki

The first paper for this lecture [1] categorizes various options for supporting speculative multithreading in hardware, while the second [2] describes a way to implement it in software without specific architectural support. The class discussion covered the requirements and tradeoffs of software, hardware, and hybrid support for speculative multithreading.

1 Paper 1: Speculative Multithreading Taxonomy

1.1 Summary

Thread-level speculation (TLS) is a way to extract parallelism from applications where it is not easy to guarantee that threads are independent. To implement TLS an architecture must buffer state, since it needs a way to back-up when a violation occurs. This paper introduces a taxonomy used to classify the approaches to buffering speculative memory state. The authors classify architectures based upon where they store their speculative memory as well as the number of speculative tasks allowed per processor.

In Architecture Main Memory (AMM), committed memory is stored in the main memory system. The paper describes two flavors of AMM, lazy and eager. In eager merging, the speculative state is merged into main memory as soon as a thread commits. The next speculative thread is not allowed to become non-speculative (the head) until this data is merged. In some cases, this merging is on the critical path of the program. Lazy merging is free of this problem since thread is allowed to pass the head token before its speculative data has been transferred to main memory.

In Future Main Memory (FMM), main memory contains the most recent version of each piece of data. If there are speculative versions of a section of memory, the committed state is kept in a buffer. In general, FMM performs commits faster than AMM since the formerly speculative data is already in the memory hierarchy. However, violations are more costly in FMM since it must restore the state of the memory hierarchy.

They also classified speculative architectures based on how many non-committed speculative tasks can be assigned to each processor. In a Single Speculative Task (SingleT) architecture, a processor cannot be assigned another speculative task until its current task is committed. In SingleT architectures, load balancing is important since a short speculative thread will cause a processor to stall until a long head thread has completed.

In contrast, Multiple Speculative Tasks (MultiT) architectures can assign another task to their processors as soon as these processors complete their current task. There are twos flavors of MultiT. In single version MultiT the processor's local memory only allows one speculative version of each piece of memory. Thus, if a speculative task needs to create its own version of a piece of data that a previous not-yet-committed speculative task already made speculative, the processor must stall. Multiple Version MultiT has no such restriction.

1.2 Conclusions

Lazy memory outperforms Eager by 30% in a multi-chip multiprocessor, 9% in a CMP. Multiple Version MultiT outperforms SingleT by 32% in a multi-chip system and 23% in a CMP. These improvements are fairly orthogonal, so there is significant benefit to employing both. Finally, Lazy AMM is competitive with FMM even though it has lower hardware complexity.

1.3 Critique

This paper does put forth a taxonomy for classifying speculative architectures. However, it does not put forth any new ideas; it merely evaluates old ones.

1.4 Future Work

Speculation is promising way to extract more parallelism from applications. Determining which type of speculation support is optimal for which applications is an open interesting questions.

2 Paper 2: Raw

The literature contains many proposals for supporting speculative multithreading in hardware for multiprocessor systems. These hardware mechanisms are mainly used for detecting dependency violations and doing rollbacks. This paper talks about supporting speculation in a CMP environment without specific hardware support. The paper describes a scheme calls Software Un-Do System (SUDS) and explains how dependency checks and rollbacks can be handled by software.

In the proposed scheme, the compiler attempts to make speculative threads out of loop iterations. It first categorizes each memory access in an iteration as one of three different types:

- Private references: Variables specific to that iteration
- Loop-carried references: Values that should be propagated to later iterations

- Ambiguous references: Locations which can not be analyzed at compile time

The compiler tries to privatize all the variables that are specific to an iteration of the loop. On the other hand, for loop-carried values, the compiler inserts direct tile-to-tile communication instructions to propagate values from one loop iteration to the next. Ambiguous references are again replaced by instructions that communicate with memory nodes.

Dependency analysis is done by a set of tiles assigned to work as "memory nodes" in the RAW CMP. These nodes also are responsible for checking addresses to see if the current location being written has been previously read by a later iteration. They also do rollbacks by keeping a log of the previous versions of variables.

The compiler analysis explained in the paper is useful not only in this scheme, but also in architectures that have specific hardware to support speculation. The software optimizations on top of hardware support will reduce the number of speculative accesses to memory and force values to be propagated for known loop-carried dependencies, which will decrease the number of potential squashes.

On the other hand, although the scheme seems not to have any specific hardware support for speculation, it relies on RAW's low-latency communication network, which allows the compiler to insert communication instructions at the register level between two tiles. Another weakness is the evaluation of SUDS; only one application is considered as a case study, and no real performance numbers are mentioned in the paper.

3 Class Discussion

3.1 Dividing responsibilities for speculation

The first question posed in the discussion was where to draw the line between hardware and software support for speculation; that is, buffering state, knowing when to squash instructions, and merging state in memory.. We also concluded that heavily optimizing in software, as Raw does, should be done in every situation because it doesn't add at all to the cost of the system. In fact, the hardware studies might have been substantially different and shown much less need for complicated support if the code had first been optimized in software.

3.1.1 Buffering state

Raw's approach to buffering state is to have one processor keep a log in case rolling back becomes necessary. Someone suggested adding hardware support for buffering state, so that the entire chunk of code wouldn't have to be rolled back.

3.1.2 Dependency detection

Since the Raw implementation does not include hardware support for this (or most anything else for that case), it requires software to take care of dependency detection through message passing. This is in opposition to using hardware techniques like having dirty bits in memory.

In order to be able to exploit TLP even with dependencies we can have staggered execution, or have some forwarding (or synchronization) mechanism.

3.1.3 Merging state

If system hardware only supports eager commits to memory, it may be possible to fool it into the more optimal policy of lazy commit. One way to do this is to either have an intelligent memory system or have one processor act as the interface to the memory. All the stores would then run through this CPU, which could keep the data structures needed for lazy commits. The Flash multiprocessor does something similar for its cache coherence.

The Raw implementation doesn't include caches, which makes purely software speculation much easier. If caches are used for non-privatized data, then some hardware support for speculation is absolutely necessary to merge state correctly.

Another idea was to use other tiles' local memory as speculative buffers for architectural main memory (AMM), instead of using future main memory (FMM). Both approaches make sense; the tradeoffs involve how close data needs to be to the processor that uses it and how many squashes are expected. FMM is optimistic that few squashes will occur, while AMM is more pessimistic.

3.2 Sources of parallelism

The Raw paper only looks for speculative threading opportunities in loops. Loops are the low-hanging fruit of speculative parallelism because they are regular and relatively load-balanced, but they are not the only opportunities for speculation.

The next obvious source of speculative parallelism is procedure calls. There are two options for speculating in this situation. The first is to execute the procedure as the "real" thread and the code following the call as the speculative thread. The second is to look ahead in the code for procedure calls and run those speculatively in parallel with the main code segment. In any event, procedure calls are only good targets for speculation if they are long enough to justify the overhead but not so long that they modify a lot of state.

It was also observed that loop-level parallelism is ideally exploited as data-level parallelism, but Raw turns everything into thread-level parallelism.

3.3 Mechanics of speculation

It is important to define "who" speculates, and what is needed to have in consideration when deciding wether to speculate or not. It is the compiler the one that marks the sections where we should speculate, as well as the size of the speculated code (there has been work that determined that the optimal size of speculated code is between 1K to 3K instructions). It is also important to observe that when we use software to speculate we can examine a larger code window than we do when using only hardware, to find opportunities to speculate and to decide wether to speculate or not.

Other questions to answer include the granularity of the speculative code. Namely, what size of memory to keep as a unit (cache-line, byte, etc.) when there is a chance we will need to undo changes. This is application dependent as well as architecture dependent (we have different granularity options if we have different hardware support).

Finally, the opportunities for speculation are very application-dependent. An example of an application where speculating is always a good idea is Database applications. The characteristics of DB apps. that make it an almost perfect candidate for speculation are the following:

- The queries are very independent, and therefore there are not many conflicts.
- It is a throughput based application.
- Everything is guaranteed to want to run.

3.4 Scalability of speculation

If designers know the limits on speculative multithreading in advance, it can help them choose the size and bandwidth of the communication network. In a CMP, they would also have the option of limiting hardware support for speculative multithreading to a specific region of the chip, thereby minimizing cost.

We tried to get an intuitive feel for the limits of speculative multithreading. A famously surprising result in probability is the birthday paradox, in which it is only necessary to have 23 people in a room to yield a 50 percent likelihood that two of them will have the same birthday. Similarly, a relatively small number of threads is enough to create memory contention problems. On the other hand, one way to increase speculative thread-level parallelism is to have a hierarchy of speculation, with speculative threads spun off from other speculative threads. This technique gets complicated, but may increase parallelism. One of the papers listed in "Further Reading" also explores the limits of speculative thread-level parallelism.

We finally decided that, as a rough number, hardware support for 16 speculative threads should be more than adequate.

3.5 Programming support for speculation

Let's now explore the different choices for programming model support for speculation.

3.5.1 No support

A clear first choice is no support at all. Just put all the effort into the compiler without programmer assistance. In this case, we have already discussed ways for the compiler to explore opportunities in loops, threads, procedure calls, etc.

3.5.2 SQL

An example of a language with support for speculation is SQL. As we discussed before, Database applications are very good candidates for speculation, and SQL provides developers with the tools to make almost everything into a transaction, and therefore, highly parallelizable.

3.5.3 Transaction-based C-like language

We also have examples of not-so-successful languages, like a transaction-based C-like languages. Although the idea might have been a good one, it is impossible to tell programers how to code! So the resistance to change might have meant the demise of this idea.

3.5.4 Transaction-based simulation laguage: Parsec

Here we have an intermediate case. Since the pool of developers and engineers using simulations is not so rooted into a fixed programing scheme, it is easier to introduce a new model. Besides this model might be closer to the events we try to simulate. Overall Parsec, a transaction-based simulation languag, has in some measure been useful to expose opportunities to speculate in TLP.

3.6 Speculative contexts

The question came uo if we can have speculative contexts in a multithreaded processor. The answer we agreed up is that yes we can, but we should look more carefully at some issues.

First of all, we need to be careful not to starve the real thread while moving forward with speculative threads. Also, we now definitely need taskIDs for the registers, since we can have multiple versions and copies on the same cache.

We also examined the number and size of speculative xontexts. Because we now have multiple context processors, we can probable reach our optimal performance with less than the "magical number" of processors, but we also now need to keep the size of the speculated code smaller.

We noted that some of these extra threads or contexts might not be doing real work. They could be, for example, pre-fetching data. But prefetching could kick out data that is still useful.

Because of all this, in multiple context processors, prediction becomes more important, and therefore we can try to pre-compute the values that the predictors need.

This discussion led us to conclude that speculative contexts and cache coherence protocols complicate each other, and that we need to be extremely careful in working with them together.

References

[1] (R) M. Garzaran, M. Prvulovic, V. Vinals, J. Llaberia, L. Rauchwerger, J. Torrellas. Tradeoffs in Buffering Memory State for Thread-Level Speculation in Multiprocessors. Proceedings of the 9th International Symposium on High Performance Computer Architecture (HPCA), Anaheim, CA, February 2003.

[2] (R) Matthew Frank, Walter Lee and Saman Amarasinghe. A Software Framework for Supporting General Purpose Applications on Raw Computation Fabrics. MIT-LCS Technical Memo MIT-LCS-TM-619, July 20, 2001.

[3] (B) M. Herlihy, J. Moss.Transactional Memory: Architectural Support for Lock-Free Data Structures. Proceedings of the 20th International Symposium on Computer Architecture (ISCA), San Diego, CA, May 1993.

[4] (F) J. T. Oplinger, D. L. Heine, and M. S. Lam. In Search of Speculative Thread-level Parallelism. Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT), October 1999.

[5] (F) J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A Scalable Approach to Thread-level Speculation. Proceedings of the 27th International Symposium on Computer Architecture (ISCA), Vancouver, BC, June 2000.