Beyond ILP II: SMT and variants

Lecture #13: Wednesday, 10 May 2000
Lecturer: Anamaya Sullery
Scribe: Ben Serebrin

1 Simultaneous MT: D. Tullsen, S. Eggers, and H. Levy

SMT, Simultaneous Multithreading, is an architecture where multiple threads from multiple programs share functional units simultaneously. This is useful because single programs have limited ILP. The architecture attempts to use all the resources of a single processor as often as possible by maximizing the level of sharing. Different threads may issue instructions during the same cycle and all the threads may share the pool of processor resources such as memory, reservation stations, and execution units during the same cycle.

Description of *horizontal* and *vertical waste*: Horizontal waste is created when some, but not all, issue slots are unused in a given cycle, due to situations such as dependencies. Vertical waste is the disuse of all issue slots in a given cycle, perhaps due to memory stalls. This is a somewhat artificial distinction, as it seems that the definitions have the potential to overlap. However, the distinction can be useful, depending on the model of execution. In one model, threads from multiple programs may issue in the same cycle, in which case horizontal waste due to dependencies may be reduced, while in the second model, threads from multiple programs must issue in different cycles, thus exposing potential horizontal waste. However, the first model may also cover vertical waste if one program issues while another is stalled on a memory dependency, so the boundary between vertical and horizontal is blurred.

Conventional MT machines do not allow multiple programs' instructions to issue in the same cycle.

1.1 Simulation Environment

The question was raised: why does this architecture not allow or need precise exceptions? No reason could be found from the paper. The paper was also unclear as to the operation of the simulator, whether it is trace or execution driven.

There are multiple possible scheduling schemes for threads: threads may be given equal shares of processor time or resources, or a priority system may be used where one thread receives all the resources it needs, and the remainder is distributed to the other threads. Both methods resulted in nearly identical performance.

The simulation used 8 Spec92 benchmarks, executed simultaneously, in 8 different sequences (which gave the programs a different mix and priority).

1.2 Figure 2 Ambiguity

The results in figure 2 showed that short floating point instructions took a large amount of processor wait time, which did not make sense in this context. The figure was somewhat unclear as a whole.

It was noted that different programs have different running profiles, so there is no single solution to improving performance: improvements to a given functional unit will provide only a small performance increase on average.

The simulator issues in-order and schedules out-of-order, and has a limited level of dynamic execution, so it has limited out-of-order ability.

Figure 2's black bar (processor busy) is confusing. It is speculated that processor busy might mean all slots are full, or all FUs are busy.

Both `eqntott` and `espresso` have substantial "short fp" unused issue cycles, but it is believed that these programs are either entirely or almost entirely integer applications.

"FP long" instructions are exclusively DIV instructions. "FP short" instructions are all other FP instructions. Multiply is the only "INT long" instruction.

`Nasa7` has the largest DTLB-derived unused issue cycle number, probably because it has a very large data set.

How do unused FUs get assigned to disuse bars in the graph? If a FU is never or rarely used by a program, it seems that it could be considered a part of the graph of unused issue cycles, which could explain the FP unit disuse for the integer applications mentioned above.

1.3 Processor issue configurations

The processor may be configured in several ways:

Fine Grain MT only one thread issues per cycle.

Full Simultaneous Issue Multiple threads can issue in one cycle.

- **Single Issue per thread** 1 instruction issues per cycle for each thread
- **Dual Issue per thread** 2 instructions issue per cycle for each thread
- **Quad Issue per thread** 4 instructions issue per cycle for each thread

The complexity of these configurations is limited by factors such as forwarding logic and the number of instructions that may issue for each thread. *Single issue* does not need to check dependences between instructions that issue in the same cycle, while *dual issue* and greater must check intra-cycle dependences.

1.4 Results

Figure 3 shows results. Why does the IPC saturate for fine grain MT? Because only one thread can issue per cycle, so horizontal waste results from limited IPC. A possible confusing factor is that if the issue window is filled with speculated (and later discarded) instructions, the results could appear better than they should.

The results show that as the number of threads is increased, the issue-slot utilization increases. Dual issue per thread is "good enough"' and simpler than larger issue widths. Simultaneous Multithreading with limited sharing is especially simple, since each thread has its own set of FUs.

The effects of shared and unshared L1 caches were investigated. With L2 and L3 shared caches, the L1 data and instruction caches were configured in all combinations of shared and private, and the results were compared for between one and eight threads. The performances converged as the number of threads increased, with private icache and shared dcache giving slightly better asymptotic results.

1.5 Wrapup

Simultaneous Multithreading vs. Multiprocessor architecture: SMT can dynamically allocate jobs to different FUs, while Multiprocessing is statically (at compile/programming time) configured, so a program written for a Multiprocessing system is less scalable to new architectures.

SMT tries to keep FUs busy, but FUs are not expensive; bandwidth is expensive.

2 Trace Processors: Rotenberg, et al

A *trace* is a sequence of executed instructions which the trace processor decides may be treated as a unit. Trace Processing uses branch and data prediction on the trace level, rather than on the instruction level; prediction within the trace is not performed. This is done in an effort to increase ILP.

The paper was unclear as to how traces are built. They may be constructed upon execution, where the first execution of some instructions is not done within a trace, or traces may be built speculatively. It is possible and may be preferred to build traces at execution and insert them into the cache when the traces are retired.

In this paper, traces are no more than 16 instructions, and performance was slightly better for lower numbers of instructions. Reexecution is *selective reissue*, where only the instructions affected by misprediction are reissued. This avoids excessive squashing. It was noted that because of the complexity of determining which instructions must be selectively reissued, it is likely that more time will be spent in the reissue check than in the process of fetching them from the trace cache, which is designed to have very high bandwidth.

The concepts of live-in and live-out data is presented: *live-in* data refers to data used within the trace that has a source dependency outside the trace, while *live-out* data is generated by the trace and is needed by another trace for input. Data which is not live-in or live-out is internal to the trace and does not affect prediction or extra-trace dependency.

Local trace prerenaming is used to avoid polluting the global rename space. This allows two useful improvements: local register files are made possible, which may have a very high bandwidth because they do not need to communicate cross-chip with all chip resources; and global renaming is moved to the trace level, rather than the instruction level. Trace-local intermediate values are also locally prerenamed. The renaming is performed serially in the trace units, rather than instruction by instruction.

How are dependencies handled with speculative execution? Mark live-ins that are affected by speculative execution and snoop the result bus; reexecute those instructions that are affected (possibly multiple times) until the thread may retire.

2.1 Trace selection

Traces are made when

- 16 instructions are read
- control changes that exit the trace are detected
- call directs (loop branches) are encountered

The trace predictor predicts the next trace for each trace encountered, using a *path history*, which is a history of traces taken or not taken. A value predictor is used to predict live-in values, using previous values and confidence estimators based on the sequence of previously-used values. This simulation used very large tables, probably larger than could be used in practice, but this simulation was performed to explore the new design's potential.

The simulation had a complex system, but only a 5-stage pipeline, probably because the authors did not care to modify Simplescalar's 5-stage pipeline.

On trace dispatch, the live-in registers look up names in the global register rename map to find the locations of their source values. Live-outs allocate a new global name.

How do traces retire and reissue? Keep completed traces in a reorder buffer and retire them in order. Do not deallocate FUs from pending traces until retirement, so when a trace must reissue due to dependency, the FUs are available.

Misprediction handling: Snoop the bus and look for mispredicted values. Reexecute if necessary. If a store has caused a pre-executed load address to change, the sequence number is checked to see if the store would have occurred before the load in an in-order execution. On control mispredictions as well as data misprediction, selective reexecution is used. It was noted that this method requires high bandwidth–can it really fetch everything it needs? However, high bandwidth is not the problem, because the instructions are still stored in the reservation stations, so no refetch is necessary. The practical problem is with the complex dependency checking for the determination of what must be reexecuated. Maybe this is not a a great solution. No results were given for this method.

2.2 Results

Figure 3 shows trace cache performance in terms of miss rate. The results are dependent on hopeful future work in data prediction.

One of the problems with using Processing Elements to contain traces is that each of the PEs must be as complicated as a single superscalar processor, to obtain only slightly increased performance over a single superscalar processor. Figure 6 provides a rough comparison of superscalar and trace processors. For a given issue width, superscalar processors generally have higher IPCs, but the paper emphasizes that comparing equal issue widths is misleading because issue width does not have full relevance for trace processors.

3 A Dynamic Multithreading Processor: H. Akkary and M. Driscoll

This paper presents a new architecture that departs from other multithreading architectures by using hardware to create multiple threads in the *same* program. This processor generates threads on the fly.

3.1 Comparison: Superscalar, Multiscalar, Simultaneous Multithreading, Dynamic Multithreading

Superscalar tries to do speculative executing using branches and traces.

Multiscalar finds parallelism using tasks, but has multiple processing units.

SMT has multiple contexts and one common resource pool.

Dynamic Multithreading is a SMT, but each thread is from the same execution unit.

3.2 Thread selection

Threads are chosen by the hardware using procedure and loop boundaries to determine the end of the thread. The paper did not say what to do in the case of nested loops. Possibilities we discussed were:

- only have one thread per inner loop

- continue speculatively on the fall through, which might be far in the future and thus could be completely invalid by the time its result is needed
- launch several iterations of the outer loop simultaneously (but how is the correct level of loop to be determined?)

Thread interdependence must be considered when launching increasingly speculative threads–we need a likelihood estimator to determine the best thread to launch next. If a thread finishes successfully, increment its associated counter to give it priority for the next time it is executed.

Trace buffers and selective recovery are shown in Figure 3. A local copy of the registers and renaming are kept so results from other threads may enter each thread, which is necessary since threads are executed with the speculation that all data values are correct. Because of this, we need to keep track of everything, and use thread ids to monitor and fix mispredictions between threads. When a misprediction is detected, the affected instructions are found and reexecuted.

It was noted in the discussion that this method serves as a wrapper for the processor to provide what may be considered a fancy prefetch service which is very good. The wrapper includes multiple fetch units (including rename units), all acting speculatively, and all units must monitor the result buffer. With all the speculative and reexecution monitoring hardware, the processor is very small compared to its support circuitry.

3.3 Results

200 instructions per thread was found to be optimal. Only 4, 6, and 8 traces were used in this test. Figure 8 shows the percentage of retired (i.e., correctly speculated) instructions, which is fairly good for all tests (in the range of 25-55%). This technique mainly acts as a new fetch organization, rather than a radical improvement in performance.