EE482A Spring Quarter 1999-2000
Project
Assigned 4/19/00
Checkpoint 1 5/1/00, Checkpoint 2 5/10/00, Due 5/24/00

You are to propose and execute a short investigative study into improving the single-thread performance of a computer system. Your project should proceed through the following phases:

1. **Concept:** generate an idea for improving performance. Some suggested ideas are listed below
2. **Hypothesis:** state an unambiguous hypothesis that can be experimentally (or analytically) proven or disproven in a short period of time.
3. **Evaluation:** conduct experiments and/or perform analysis to prove or disprove your hypothesis.
4. **Description:** write up your findings and your methodology

You are encouraged to work in groups of up to five students on this project.

To give you feedback with the project, and to encourage you to get an early start, there are two project checkpoints due. For checkpoint 1, due on May 1st you should have completed your concept, have a clear hypothesis, and a proposed method for evaluation. For checkpoint 2, due on May 10th, you should be midway into your evaluation with any experimental software at least partially working and some preliminary results. You need not turn anything in for the checkpoints, but you should come to office hours or make an appointment to review your progress with Prof. Dally.

Suggested Concepts

The following concepts are intended as suggestions. You are welcome and encouraged to generate topics that are not on this list, or to modify these to suit your needs. The concepts are ranked according to difficulty. Unless you have a very large amount of time to devote to this, I suggest you pick one of the easier ones.

Follow on research:

A good source of concepts is our set of required and recommended readings. Many of these papers leave questions unanswered that would make good topics for a short study.

1. **Trace cache experiments.** Add a trace cache to the simple-scalar simulator and experiment with fill and replacement policies. Can you develop fill and replacement policies that outperform the standard policies. Alternatively you could experiment with block prediction mechanisms.
2. **A better branch predictor.** This area has been pretty well picked over but there's always a chance you can come up with a better method.
3. **Hints to the cache/memory system.** Consider annotating loads with information about whether a line should be fetched normally or initiate a stream. How would the affect memory performance? What other information can you encode about each load (e.g., latency tolerance, miss probability) and how would you exploit it?

4. **More sophisticated stream buffer.** Explore stream buffers that avoid polluting the cache, that can handle arbitrary stride streams (and perhaps even indirect accesses) and avoid prefetching on data loads that aren't streams.

Short Topics

1. **Power efficient processor architecture.** Evaluate some architecture concept: front-end, memory system, or execution core against a combined power and performance metric rather than a performance-only metric. How do tradeoffs change when power is also an issue?
2. **Wire efficient architecture.** Investigate methods that will allow super-scalar processors to continue to scale in the presence of slow on-chip interconnect. For example, how can multiple execution units be controlled to keep data values *local*.
3. **Reconfigurable architecture.** Explore uses for reconfigurable arithmetic units, for example to subsume sequences of normal instructions.
4. **Reduce branch misprediction penalty.** Explore ways to reduce branch misprediction penalty, for example by breaking branches into to or three parts, by using caches of various types, etc....
5. **Determine how much ILP there is.** For some program or programs, determine the limits of performance by suspending reality in terms of practical limits on execution parallelism, prediction accuracy, etc....

Very Ambitious

1. **Instruction set design for high-bandwidth issue**. Is it possible to design an instruction set, perhaps a VLIW instruction set, that achieves very high issue performance, enough to satisfy an 8-way to 16-way execution unit, without resorting to the complexity of trace caches and eager execution? One possibility is to use a compiler to pack the most likely set of instructions into fixed-sized blocks, perhaps 16 instructions wide. Also encoded into each block would be some form of explicit predication, to conditionally disable some of the instructions in the block. The encoding should allow multiple exit points from each block. Also the branch predictor should be exposed to the compiler so they can work cooperatively. This is just one possibility. Many more opportunities open up when the compiler, the ISA, and the hardware are all fair game.
2. **Scheduling instruction blocks.** Modern superscalar processors schedule instructions for execution one at a time. Is it possible to increase efficiency by scheduling groups of related instructions together? For example, rather than reading each instruction into a reservation station and enabling it independently, an entire code block of related instructions could occupy a single reservation station and be scheduled as a unit. In particular, instructions could be broken into such scheduling blocks only at points, like loads, where timing is difficult to predict at compile time.
3. **Exposing the memory hierarchy.** As DRAM latencies increase beyond 100 CPU cycles (400 issue slots on a 4-way machine), a very large number of outstanding references to DRAM is needed to maintain high bandwidth on codes that get poor spatial and temporal locality. One way of addressing this problem is to add a fast memory of moderate size (say 1MB) explicitly to the hierarchy. Instead of prefetching to the cache, data is explicitly staged through this memory. This can win in two ways. First, since there is no run-time allocation, an essentially unlimited number of staging references can

be simultaneously outstanding to load data into this staging buffer. Second, much data is produced and then consumed a short time later. This data never needs to be moved out of this buffer, saving memory bandwidth that would otherwise be used in a typical cache-based hierarchy. A less radical approach in the same direction is to give more control over the cache to the ISA.

4. **Optimizing the execution unit for locality.** As semiconductor technology advances, wires are becoming more of a bottleneck in terms of delay, area, and power. At the same time, larger numbers of function units require more wiring for issue and result forwarding. There are many approaches to reducing this communication burden. The complete connection provided in most systems today can be replaced with a more efficient interconnect. This interconnect can then be exploited by allowing the compiler (or run-time hardware) to place operations on nearby function units. Other optimizations are also possible.