Project Proposal
Cody Martin and Adu Bhandaru

1. Problem

The memory-bottleneck is well known problem in computer architecture. In general, we try to alleviate contention for memory, the bus, and data using caching, smart request scheduling, and many other techniques. Many of these methods maximize system throughput by providing higher quality of service and hiding more latency.

Caching is the bread and butter method for exploiting spacial and temporal locality in an application's access patterns. There have been many studies done on exploiting regular access patterns (prefetching, for example). There are also some very compelling ideas on how to hide latency for irregular accesses as well (run ahead execution, among others).

However, we believe we may be able to exploit some of these regularities in memory access in addition to just spacial and temporal patterns. As of now, cache configurations are rigid and cannot exploit certain predictable repetition in memory access patterns. Having a dynamic cache structure that tailors itself towards the currently executing application could yield higher performance by removing a great deal of cache conflict misses that would normally occur with a statically configured cache.

2. Novelty

People have tried to tackle this problem with prefetchers, runahead execution, and general purpose access pattern predictors to hide latency. With configurable caching, we can certainly augment these techniques, if not replace them.

Currently, many microarchitecture designers use a branch predictor that implements hysteresis to determine what prediction to use during execution of each branch in the code. We can use this same concept of hysteresis to waiver between two or more cache implementations. This is an idea that we believe to be novel in the space, but uses techniques that have been used in the past to assure effectiveness.

3. Idea

Consider having the private L1 cache be configurable. That is, design it such that we can dynamically determine the optimal number of sets (and therefore associativity) and perhaps even the block size. Imagine implementing the selection logic for a particular block for a number of different configurations and connecting them to the same cache. At any given time only one scheme would be enabled.

The which configuration to enable could be specified either by the programmer (needs ISA exposure) or determined dynamically. If programmatically, the programmer could just choose the configuration that best suits the application. Alternatively, we could record the access patterns generated for a particular core and determine the optimal cache configuration ourselves in hardware.

Even if only two configurations were implemented, we could still achieve good performance gains. Consider the graph of associativity to number of conflict misses. If we use a saturating counter for hysteresis, you could imagine a simple mechanism for choosing which configuration to use. Even just changing to a two-way cache makes a big difference in handling requests. Further, configurable block size from just 4-byte to 8-byte could make the difference between containing a full object or only half of it.

4. Hypothesis

By having the cache configuration adapt to the needs of access pattern, we can reduce the occurrence of conflict misses. Of course, we will still experience both compulsory and capacity misses. Further, this technique can be applied to systems that already employ runahead execution etc., and we should still realize performance gains.

We expect the benefits of dynamic cache configuration to manifest themselves in performance in a number of ways. Consider a program that accesses address **A** 50% of the time, and addresses **B** and **C** 25% of the time, where they all map to the same set. Not only does such an access pattern incur a large number of conflict misses in a direct-mapped cache, but it also has very poor cache space utilization. By recognizing this dynamically, we can switch to a more associative cache. Such a configuration would offer much better utilization.

On the other hand, consider a program that iterates over a small array repeatedly. For the purpose of the example, assume the array is smaller than the capacity of the cache. Here, a direct mapped cache performs very well. Not only is it very low latency, but also has excellent space utilization. A more associative cache would only have one line used in each set, assuming the array had elements the same size as the blocks.

As demonstrated, dynamically picking a cache configuration can keep more of an applications working memory set in the cache. This also means preventing costly evictions and better cache warm up times.

5. Methodology

Since we do not yet know if this is even possible in design practice, we will need to first determine whether or not we

are able to configure caches dynamically. We have realized that this design may incur high hardware costs, and it will be necessary for us to determine the feasibility of this idea as well as weigh it against the hardware costs to decide whether or not it is a worthwhile endeavor.

To evaluate feasibility and the hardware footprint, we will need to investigate modern day SRAM structure. This includes the bank structure, selection logic, and interconnect with the processor and the L2 cache. Possible areas for concern include ensuring block ordering is consistent with cache configuration, how long a reordering would take, etc.

In addition, we need to simulate the additional capacitance the extra bit-lines will put on the circuit, and how strongly this will impact latency. We will have to explore various latency modeling techniques for this.

Finally, once we have a convincing micro architecture specification, and have implemented a cycle accurate simulator that includes our configurable cache, we would like to benchmark testing. Specifically, we will run a standard (or well established) simulator on series of well known benchmark programs. Then we will run a version of the simulator with the configurable cache (as described) and compare the performance.

6. Plan

The following list are some general goals we wish to meet as we conduct our research.

- Milestone 1: Do some background research on other flavors or variants of configurable caches. Perhaps this idea has been explored and we have not found the paper. Determine feasibility of designing more complicated selection logic, estimate hardware costs incurred. If there are no red-flags we will move to milestone 2. A red-flag would be something like being unable to wire up two-types of selection logic to the SRAM banks, or incur very high latencies in doing so.

- Milestone 2: Specify and possibly implement the microarchitecture in Verilog and estimate the latencies for all operations. Specifically, investigate if we can return L1 hits in the same cycle, or if we need 2 cycles (or something) for a particular configuration.

- Milestone 3: Augment the simulator with configurable caches for each core, ensuring the microarchitectural specifications are met with regards to timing. Run the simulator on a well known benchmark suite.

- Final Report: Analyze the results from the benchmark tests. Observe performance gains (or loss) and speculate as to why certain tests performed better or worse. Determine where we can make improvements and log them.

The number of configurations we support will rely on a number of things. Time permitting, we would certainly like to explore the possibility of dynamically choosing between as many as 4 different configurations. However, a more reasonable goal would be to fully implement and test a two-way configurable cache. Second, if our initial analysis indicates the hardware overhead is too great to support additional configurations, we will not pursue that route.