Using a Serial Cache for

Energy Efficient Instruction Fetching

Glenn Reinman[†] Brad Calder[‡]

[†]Computer Science Department, University of California, Los Angeles

[‡]Department of Computer Science and Engineering, University of California, San Diego

Abstract

The design of a high performance fetch architecture can be challenging due to poor interconnect scaling and energy concerns. Way prediction has been presented as one means of scaling the fetch engine to shorter cycle times, while providing energy efficient instruction cache accesses. However, way prediction requires additional complexity to handle mispredictions.

In this paper, we examine a high-bandwidth fetch architecture augmented with an instruction cache way predictor. We compare the performance and energy efficiency of this architecture to both a serial access cache and a parallel access cache. Our results show that a serial fetch architecture achieves approximately the same energy reduction and performance as way prediction architectures, without the added structures and recovery complexity needed for way prediction.

1 Introduction

The performance of any architecture is limited by the amount of instruction fetch bandwidth that can be supplied to the execution core. Instruction cache performance is a vital part of achieving high fetch bandwidth. An energy efficient fetch design that still achieves high performance is also important because overall chip energy consumption may limit not only what can be integrated onto a chip, but also how fast the chip can be clocked [7]. Brooks et al. [1] report that instruction fetch and the branch target buffer are responsible for 22.2% and 4.7% respectively of power consumed by the Intel Pentium Pro. Brooks also reports that caches comprise 16.1% of the power consumed by the Alpha 21264. Montanaro et al. [6] found that the instruction cache consumes 27% of power in their StrongARM 110 processor.

Set-associative cache designs can improve performance over a direct mapped cache by reducing thrashing among cache blocks that map to the same cache index (i.e. among all ways within a cache set). This extra associativity comes at the price of increased energy. During a parallel cache access, both the tag and data components of all cache ways (blocks) in a given cache set (index) must be driven. If the tag component of one of the ways matches the desired address, then the corresponding data component of that way is selected to be output. But regardless of which way matches the desired address, all ways in the set are driven on the bitlines of the cache to the logic that selects a single cache block to output.

Way prediction [4, 13, 9] has been proposed as a means to provide low-latency, energy efficient cache access. Way prediction has been used in a number of real world architectures, including the Alpha 21264 [10], which makes use of the Next Line and Set (NLS) [3] predictor, a branch predictor with integrated way prediction. However, way prediction requires additional hardware to perform the actual way prediction, verify the correctness of a prediction, and recover in the event of a misprediction.

In this paper, we compare the performance of using way prediction [4, 13, 9, 10, 3] to using a serial

cache for instruction fetch. A serial instruction cache separates its tag and data lookup into separate cycles, so only the correct *way* needs to be driven. We show that this achieves the same energy savings as way prediction, with only a 1% loss in performance when compared to way prediction fetch architectures.

First, we present a way prediction architecture for a high bandwidth decoupled fetch architecture, where multiple way predictions are provided per cycle. In comparison, prior way prediction architectures are coupled to the instruction cache, providing a single way prediction per cycle. Then, we compare the performance of the serial cache architecture to the NLS style of way prediction used in Alpha processors, as well as the multiple way prediction architecture.

Since we examine results using a processor simulator, we arrive at a different conclusion than prior serial instruction fetch research by Inoue et al. [9]. They examined instruction cache miss rates and concluded that way prediction was advantageous over a serial cache design due to the additional pipeline stage it introduces from serializing the tag and data access. Our results show that for a moderately pipelined architecture (8 cycles from fetch to execute), the performance loss of using a serial design over way prediction is only 1% on average.

Section 2 presents the serial fetch architecture and the way prediction architecture we evaluate in this study. Section 3 covers the methodology of our simulations. Section 4 presents results for these architectures and we conclude in Section 5.

2 Fetch Architectures

In this section, we outline two different instruction cache configurations for use with our high-bandwidth, decoupled front-end architecture.

2.1 The Decoupled Front-End

In our prior work we explored an architecture that decoupled the branch prediction architecture from the instruction fetch unit (including the instruction cache). The branch predictor and instruction fetch unit are separated by a queue of fetch addresses (branch predictions) called the *Fetch Target Queue* (FTQ) [15]. The FTQ has two primary functions, it provides latency tolerance between the branch prediction architecture and the instruction fetch unit, and it provides a glimpse at the future fetch stream of the processor.

The ability of the FTQ to tolerate latency between the branch prediction architecture and instruction cache enables a multilevel branch predictor hierarchy called the Fetch Target Buffer (FTB) [14]. The FTB combines a small first level predictor that scales well to future technology sizes with a larger, pipelined second level structure, which provides the capacity needed for accurate branch prediction. With sufficient branch predictions stored in the FTQ, the architecture is able to tolerate the latency of the second level branch predictor access while the instruction fetch unit continues consuming predictions already stored in the FTQ.

This decoupled design provides us with great flexibility in the selection of an instruction cache. We are able to pipeline the instruction cache, without impacting the branch prediction architecture. Moreover, our design allows us to easily scale the number of ports on the instruction cache to provide more instruction fetch bandwidth.

2.2 Instruction Cache Design

Instruction cache performance is vital to the processor pipeline. Associativity is a useful technique to improve cache performance by reducing conflict misses in the cache. The conventional set-associative

cache design probes the tag and data components of the cache in parallel to reduce the cache access time. However, this approach wastes energy in the bitlines and sense amps of the cache as it must drive all associative ways of the data component on every access, hit or miss. We refer to this design as a *parallel* cache.

One alternative to this is to access the tag component of the cache first to determine what associative way of the data component should be driven. Such a cache is commonly known as a *serial* cache. This design has been used for L2 caches, and recently for data caches on graphics cards [11, 8]. The Alpha 21264 [10] splits the tag and data component of its direct-mapped second level cache, effectively creating a serial L2 cache design.

Figure 1 shows one design of a serial cache that we call the *multicomponent (MC) serial cache*. This cache has the same tag component arrangement as a regular set-associative instruction cache, but rather than a single set-associative data component, there are a number of direct mapped data components. The 16KB 2-way associative configuration shown in Figure 1 has two direct mapped data components, each only 8KB in size. A 16KB 4-way set associative MC cache would have four 4KB direct mapped data components. Each direct mapped data component has its own decoder, sense amps, and other auxiliary structures. At most one data component is enabled at each access, depending on tag information.

2.2.1 Serial Fetch Architecture

Figure 2 presents the *Serial Fetch* architecture using the MC cache. The Serial Fetch architecture performs its tag and data lookups in separate stages of the pipeline in order to save energy. If there is a hit in the tag stage, then only the correct direct mapped data component is accessed. If there is a miss, then no data component is accessed, and the data is brought in from the next level of the cache hierarchy. This determination is entirely nonspeculative: the tag access is a fully associative lookup to determine

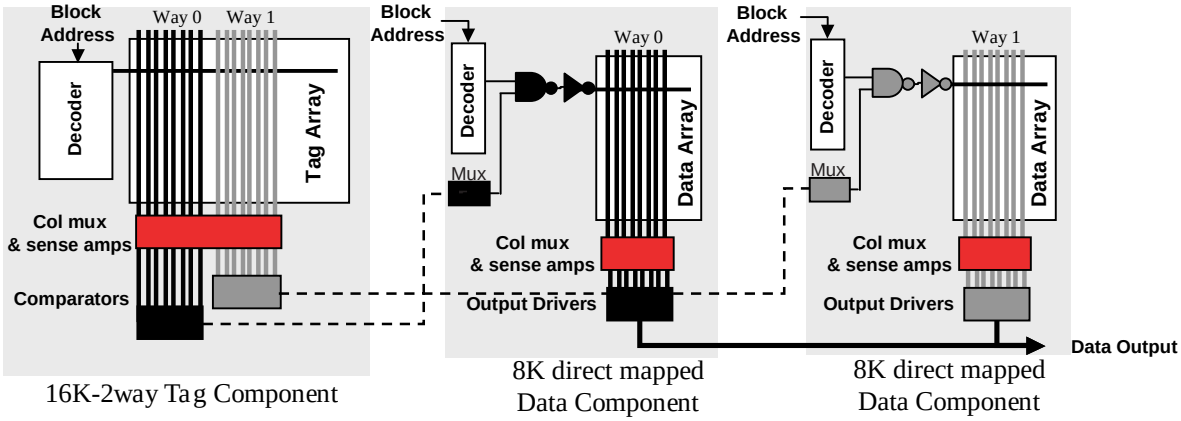

Figure 1: A 16KB 2-way set-associative multi-component (MC) cache. This has the same tag component as the instruction cache, but with multiple data components. Each data component is a direct mapped cache. For a cache of size C that is A-way set associative, there are A direct mapped caches of size $\frac{C}{A}$ forming the data components.

where the data is located. This provides significant energy savings, especially when considering caches of higher associativities.

In recent work [16], we explored selectively accessing cache ways using a decoupled MC cache to create an energy efficient *instruction prefetch architecture*. In this submission, we expand on that research by (1) examining the use of a serial cache design just for instruction fetch, and (2) compare our serial fetch design to way predicted fetch architectures. These designs and their analysis are contributions over the work in [16].

2.2.2 Next Line and Set Prediction

An existing branch prediction architecture that uses an energy efficient cache was proposed by Calder and Grunwald [3]. Their *Next cache Line and Set* (NLS) prediction architecture predicts an index into the instruction cache rather than a branch target address. This predictor provides a pointer into the instruction cache, indicating the branch target to resume fetch. This form of predictor is used in the Alpha 21264 [10]. It supplies a prediction of what associative way the predicted cache block is located

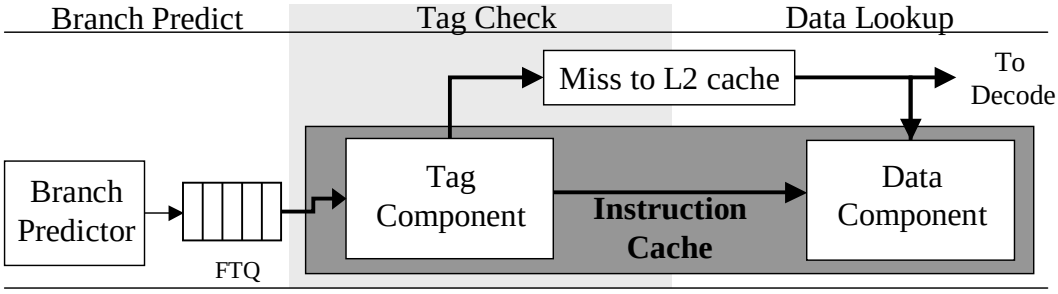

Figure 2: The base pipeline for the Serial Fetch front-end. The pipeline consists of three stages: branch prediction, tag check, and data lookup stage. The branch prediction architecture fills the fetch target queue (FTQ) with fetch addresses in the first stage. In the second stage, the MC instruction cache performs an associative lookup in the tag component of the cache to determine what direct mapped data component contains the desired cache block, if any. In the final stage, the cache block is read from the data component of the MC cache indicated in the second stage. In the event of a cache miss, the cache block is brought in from the L2 cache.

in. The authors refer to way prediction as set prediction in [3], but essentially the predictor determines what associative way contains the desired cache block. This way prediction, combined with an energy efficient cache design (like an MC cache), allows a single cache way to be selectively enabled, and can provide significant energy savings. They examined NLS in terms of reducing the access time for the instruction cache, but not in terms of its energy efficiency.

In this paper, we implemented an NLS predictor that provides one cache block per cycle, and compare its performance to the serial cache design and to a generalized way predictor (described in the next section).

2.2.3 Way Prediction for High Fetch Bandwidth Architecture

One alternative to tolerating the latency of a serial cache access is to use way prediction, as done with the NLS predictor. Figure 3 demonstrates the addition of a multi-block way predictor to the high bandwidth FTB architecture [14]. To provide a high fetch bandwidth architecture with way prediction, we first need the means to compress the tag check and data lookup stages of the instruction cache access into

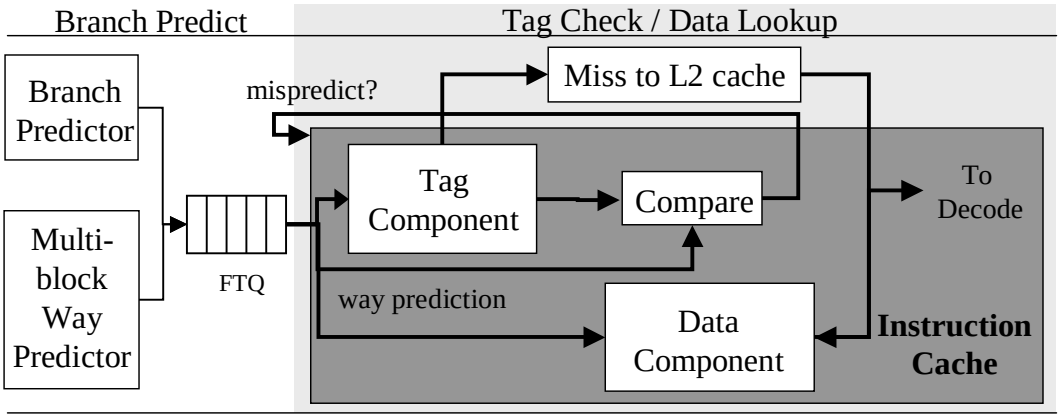

Figure 3: The base pipeline for the MC cache front-end augmented with way prediction. The pipeline consists of two stages: branch prediction and a combined tag check and data lookup stage.

a single stage – while still using the MC cache. Calder et al. [4] have suggested using a separate way predictor with a data cache to perform such an access in parallel. The way predictor that we use in this paper is a simple last n-bit state counter. Since each FTB prediction can potentially span up to 5 cache blocks (due to the default fetch distance stored in the FTB), we would like to provide up to 5 way predictions per cycle. Our multiple way predictor is direct mapped and indexed by the branch prediction fetch PC. Each cycle, 5 sequential predictions are read out of an entry in the way predictor. Our tagless predictor has 32K entries, and each entry has 5 2-bit predictors. The 2-bit counter is used to provide hysteresis for predicting the given way, since there can be conflicts in the table. The way predictor and FTB are accessed in parallel, and the way prediction is stored in the FTQ until it can be consumed by the instruction cache. One alternative to the use of a separate structure is to place the way prediction directly in the FTB, trading the additional area required by the way predictor for the access time impact that would result from widening the FTB.

In Figure 3, the tag component of the instruction cache grabs the current cache block to be fetched from the FTQ. The way prediction is consumed by a data component and the way compare hardware. The data component selected by the way predictor will drive the cache block corresponding to the

fetch prediction. The tag component searches all cache ways of the line corresponding to the current cache block. The way compare hardware will determine whether or not the way indicated by the tag component matches the way prediction (*i.e.* whether or not the correct data component was enabled). If a misprediction has occurred, the correct data component will be accessed in the following cycle (determined via the tag component access). If the way prediction was correct, the instruction cache access will have only taken a single cycle. On a tag miss, the block is brought in from the L2 cache.

2.3 Prior Way Prediction Research

Inoue et al. [9] examined using an MRU algorithm to predict what way of an associative cache to access. They also compared their results to a serial access cache (what they refer to as a phased cache). They provided results using a cache simulator, not a processor simulator, and therefore did not examine the impact of branch prediction, cache pipelining, or fetch bandwidth on their architecture. They also did not address the impact of complexity on the way prediction architecture. Finally, their way predictor, based on a MRU counter, provides only a single way prediction each cycle. They concluded from their cache simulation results that way prediction architecture was advantageous over the serial approach. Our results show that is not the case when examining results from a detailed processor simulation.

Solomon et al. [18] examined the use of a serial cache along with their Micro-Operation Cache, but did not compare the performance of such a cache to a parallel or way predicted cache configuration, which is the focus of our paper. The main thrust of their study was the evaluation of the Micro-Operation Cache, and they provide no results on the relative performance of their serial cache organization to other cache structures.

Powell et al. [13] have also examined the impact of way prediction on the energy consumption of the

instruction cache, making use of a conventional, coupled branch prediction architecture. Their branch prediction architecture is similar to the NLS predictor in that it also only provides a single way prediction (cache block) per cycle, whereas the way prediction architecture we examine in this paper can perform multiple way predictions (fetch multiple blocks) per cycle. They associate a way prediction with the PC of the previous cache address to account for the branch predictor/instruction cache coupling. In comparison, the way prediction architecture shown in Figure 3 features a way predictor that can scale to match the bandwidth of a more aggressive fetch architecture, without sacrificing predictor accuracy. The decoupled design also provides more flexibility in the design of such a way predictor (as in our decoupled branch predictor [14]), and could even provide an opportunity to check mispredicted ways using idle cache tag ports (but this is not explored in this paper). A decoupled architecture is also able to take advantage of way misprediction stalls, as it provides an opportunity for the branch prediction architecture to continue ahead of the instruction cache.

3 Methodology

The simulator used in this study was derived from the SimpleScalar/Alpha 3.0 tool set [2], a suite of functional and timing simulation tools for the Alpha AXP ISA. The timing simulator executes only user-level instructions, performing a detailed timing simulation of an aggressive 8-way dynamically scheduled microprocessor with two levels of instruction and data cache memory. Simulation is execution-driven, including execution down any speculative path until the detection of a fault, TLB miss, or branch misprediction.

To perform our evaluation, we collected results for 19 of the SPEC2000 benchmarks (selected at random). The programs were compiled on a DEC Alpha AXP-21164 processor using the DEC C and

program	16K Cache 2way IPC	MR
ammp	0.152	0.2
applu	1.082	2.4
apsi	1.266	1.1
art	0.653	0.0
crafty	1.113	18.4
eon	1.487	11.5
equake	2.173	7.1
facerec	2.643	0.0
galgel	2.889	0.0
gcc	1.072	3.7
lucas	1.009	0.0
mcf	3.062	1.2
mesa	2.332	2.8
mgrid	1.511	0.0
parser	1.497	0.2
perlbmk	1.763	5.3
swim	0.955	0.0
vortex	1.085	22.3
wupwise	2.413	0.0

Table 1: Program statistics for the baseline architecture. For a 16K 2-way associative cache, the IPC and instruction cache miss rate (%) for each benchmark are shown.

C++ compilers under OSF/1 V4.0 operating system using full compiler optimization (`-O4 -ifo`). For each benchmark, two billion instructions were executed (fast forwarded) before actual simulation. We report results for simulating each program for 200 million instructions after fast forwarding. In all cases, we use the reference data sets to simulate results. Table 1 shows the benchmarks used in this study.

3.1 Baseline Architecture

Our baseline simulation configuration models a next generation out-of-order processor microarchitecture. We've selected parameters to capture underlying trends in microarchitectural design. The processor has a large window of execution; it can fetch up to 8 instructions per cycle. It has a 128 entry

re-order buffer with a 32 entry load/store buffer. We simulated perfect memory disambiguation (perfect Store Sets [5]). Therefore, a load only waits on a store it is truly data dependent upon. To compensate for the added complexity of disambiguating loads and stores in a large execution window, we increased the store forward latency to 3 cycles.

There is an 8 cycle minimum branch misprediction penalty. The processor has 8 integer ALU units, 4 load/store units, 2 FP adders, 2 integer MULT/DIV, and 2 FP MULT/DIV. The functional unit latencies are: Int ALU 1 cycle, Int MULT 7 cycles, Int DIV 12 cycles, FP ALU 4 cycles, FP MULT 4 cycles, and FP DIV 12 cycles. All functional units are fully pipelined allowing a new instruction to initiate execution each cycle.

We use a 128 entry 4-way associative FTB with a 2K entry 4-way associative second level FTB. Each fetch block stored in the FTB can span up to five sequential cache blocks. We use the McFarling bi-modal gshare predictor [12], with an 8K entry gshare table and a 64 entry return address stack in combination with the FTB. We use a 32 entry FTQ in conjunction with the FTB.

3.2 Memory Hierarchy

We rewrote the memory hierarchy in SimpleScalar to model bus occupancy, bandwidth, and pipelining of the second level cache and main memory. This study makes use of a 32KB 4-way set associative data cache and a 16KB 2-way set associative instruction cache (each with 32-byte lines). Both caches are dual-ported.

The second level cache is a unified 1 MB 4-way set associative pipelined L2 cache with 64-byte lines. The L2 hit latency is 12 cycles, and the round-trip cost to memory is 100 cycles. The L2 cache has only a single port. The L2 cache is pipelined to allow a new request every 4 cycles, so the L2 bus

can transfer 8 bytes/cycle. The L2 bus is shared between instruction cache block requests and data cache block requests.

3.3 Energy Model

The energy data we need to generate results is gathered using the CACTI cache model version 2.0 developed by Reinman and Jouppi [17]. CACTI 2.0 contains a detailed model of the wire and transistor structure of on-chip memories, verified by hspice. We modified CACTI 2.0 to model the timing and energy consumption of the front-end structures of our architecture. CACTI 2.0 uses data from $0.80\mu m$ process technology and can then scale timing data by a constant factor to generate timings for other process technology sizes. We examine timings for the $0.10\mu m$ process technology size, which makes use of a $1.1V$ *Vdd*.

CACTI 2.0 reports energy data for successful cache accesses. We modified CACTI 2.0 to report energy data for successful accesses, misses, tag probes, and writes. We further modified CACTI 2.0 to estimate the power consumption of all front-end structures, including the FTB, FTQ, instruction cache, L2 cache (a unified cache – but we only counted power from instruction cache misses, not from data cache misses), and other auxiliary structures. For each, we modified the BITOUT, ADDRESS BITS, and block size parameters appropriately. Then we track the number of hits, misses, tag probes, and writes to these structures in SimpleScalar to compute the overall energy dissipation for these structures. When we report energy dissipation results in Joules, this includes the power dissipated by *all the above listed front-end structures*.

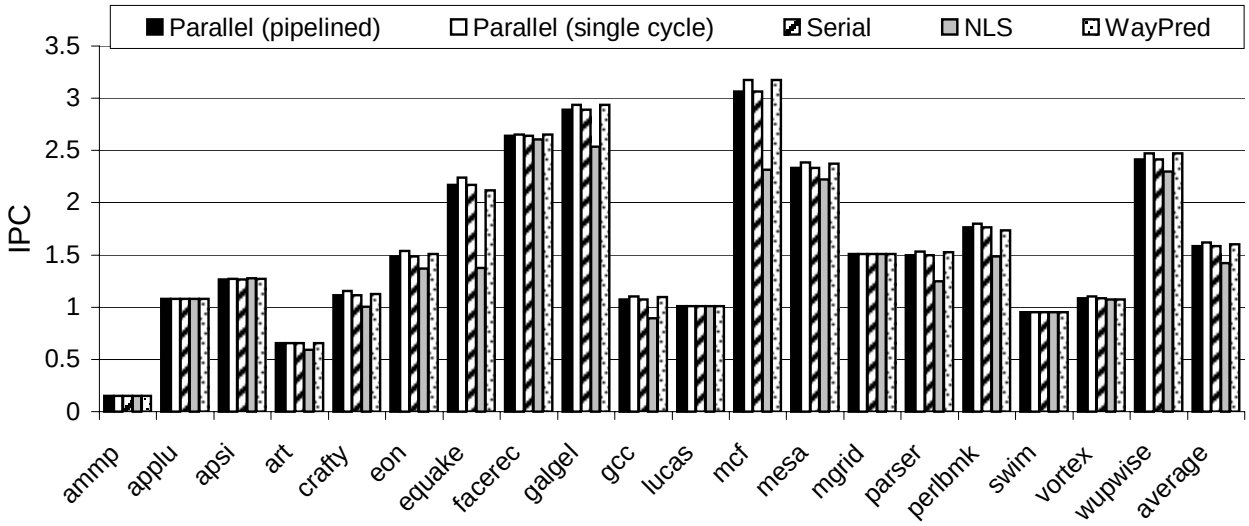

Figure 4: IPC results for five architectures: a Parallel cache pipelined over two cycles and a parallel cache with a single cycle access, both using a 2-level FTB; the Serial MC instruction cache with a 2-level FTB; the NLS architecture that provides a single cache block each cycle using an MC cache; and a way prediction architecture using an MC instruction cache with a 2-level FTB.

4 Results

Figure 4 shows IPC results for five architectures. All of the architectures but the NLS configuration make use of a high bandwidth fetch architecture (2-level FTB with a 128-entry first level). The first two bars, Parallel (pipelined) and Parallel (single cycle), use a 16KB 2-way set-associative parallel instruction cache. The parallel cache in the first bar, shows results if the cache would need to be pipelined over 2 cycles to meet a lower cycle time. The parallel cache in the second bar, shows results for a single cycle parallel cache access, if the cycle time of the cache was not an issue.

The remaining bars in Figure 4 use 16KB 2-way set associative MC serial caches. The third bar, Serial, represents an architecture with the serial fetch architecture from Section 2.2.1, where the tag and data components are in separate pipeline stages. The next two bars show results for the NLS and Way Prediction architectures. Both of these designs have a single cycle instruction cache access on a correct prediction, and a two cycle access on an incorrect prediction. So, on an incorrect way prediction, the Serial, NLS, and Way Prediction architectures take 2 cycles.

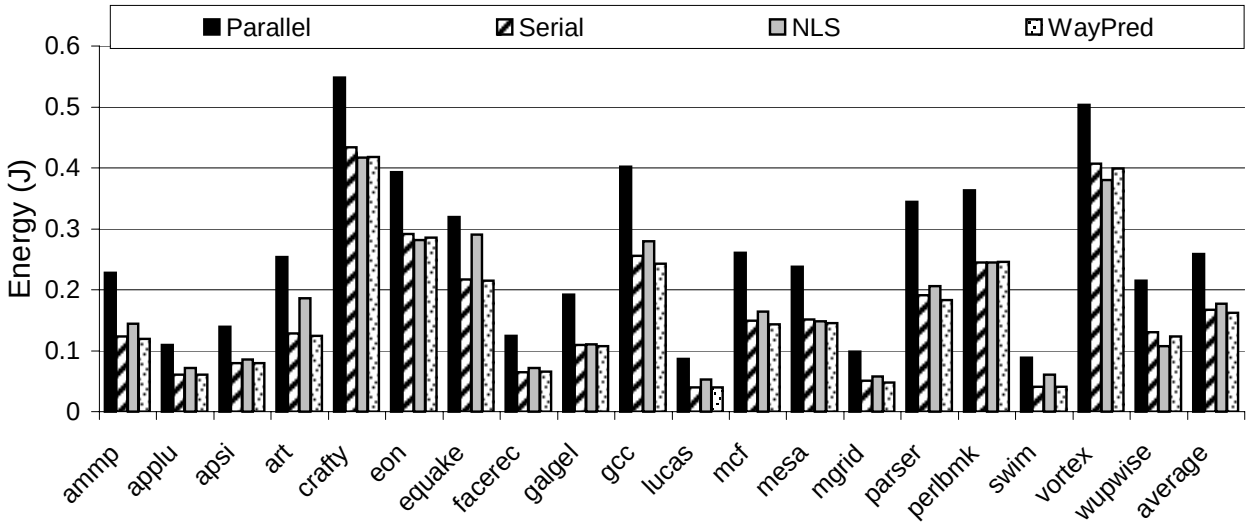

Figure 5: Energy results for four architectures: the parallel cache, the Serial cache architecture, the NLS architecture and a high bandwidth fetch architecture with Way Prediction.

Figure 5 provides energy results in Joules for four of the five architectures in Figure 4. For brevity we only show the pipelined parallel cache configuration. This Figure reports energy results for all of the major structures of the front-end only, as outlined in Section 3.3. The energy efficient MC cache expends 35% less energy on average than the parallel cache (measured relative to the Serial configuration).

The NLS architecture is the worst performer in Figure 4, as it only fetches one cache block per cycle. The other fetch architectures examined can fetch up to two cache blocks per cycle, and have two ports on their instruction caches. For some configurations, the NLS architecture uses significantly more energy than the other configurations. These benchmarks have a lower prediction accuracy on the NLS architecture.

The Way Prediction architecture is able to slightly outperform the Parallel and Serial architecture on average, due to its single cycle instruction cache access. Its performance is very close to that of the Parallel (single cycle) architecture. The Way Prediction architecture suffers a single cycle stall when the way is mispredicted, but the loss in performance is small. The relative performance of a given benchmark on Serial and Way Prediction will depend on whether branch mispredictions or way

mispredictions are more prevalent. A benchmark like `equake`, which has a relatively low branch misprediction rate and a relatively high way misprediction rate actually sees better performance from the Serial architecture. The opposite is true of a benchmark like `galgel`, which has an extremely low number of way mispredictions. The way predictor achieves nearly 95% accuracy on average for the benchmarks we examined.

Figure 6 shows the results for perfect way prediction (i.e. no mispredictions) compared to serial and the way prediction architecture, all for a smaller, but more associative instruction cache. Results show that perfect way prediction provides only a 1% improvement in performance over the way prediction results.

It is interesting to note that a serial instruction cache actually has the potential to outperform a way predicted instruction cache (i.e. in the case of `equake`). On a way misprediction, the WayPred architecture will also take 2 cycles to access the instruction cache, but it will have wasted a cycle (and a data component cache port) on a mispredicted access. If mispredictions are frequent and branch mispredictions are infrequent, then a serial instruction cache can outperform the more complex design of a way predicted instruction cache.

Inoue et al. [9] found similar way prediction accuracy for the instruction cache, reporting an average 96% accuracy for the benchmarks they examined. They concluded that using a way prediction architecture was advantageous over the serial approach, since the serial architecture would introduce an additional pipeline stage to provide the tag and data component serialization. They only examined the performance of serial and way prediction in terms of miss rates. In comparison, our results show that adding the additional pipeline stage for the serial fetch architecture degrades performance by less than 1% in comparison with the way prediction architectures. These results assume an 8 cycle minimum branch misprediction penalty, which is a very conservative misprediction penalty considering the

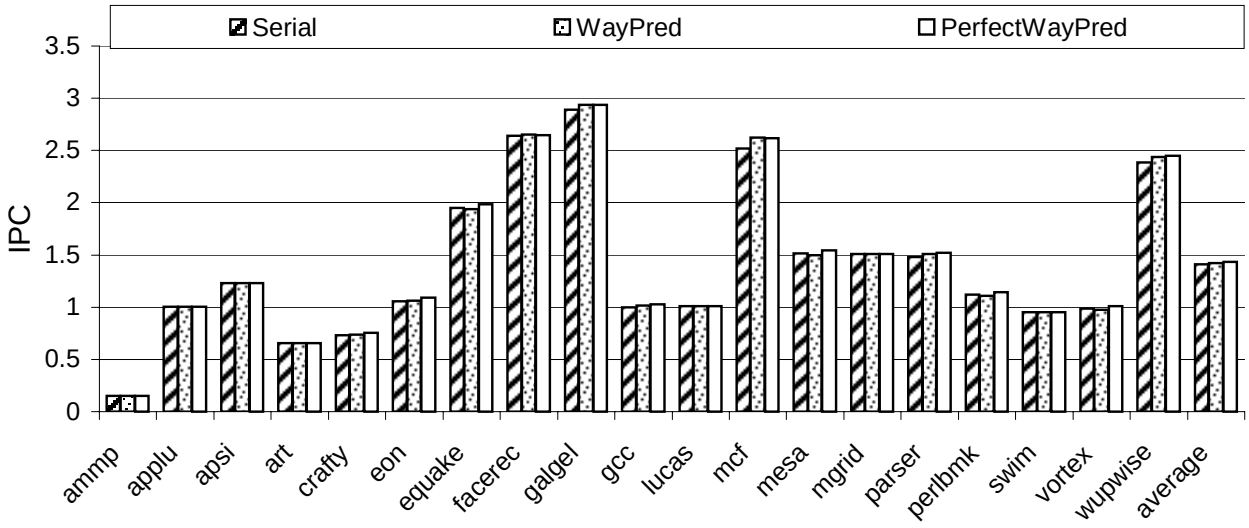

Figure 6: IPC results for three architectures, all with 8KB 4-way set associative MC instruction caches and 2-level FTBs: a serial instruction cache configuration, a way predicted instruction cache configuration, and a perfect way predicted instruction cache configuration (i.e. instruction cache access is always a single cycle).

pipeline depth of current and future processors. As the pipeline depth increases, the IPC difference between serial and way prediction will be even less.

Although the Serial and Way Prediction architectures have similar energy savings and performance, the Way Prediction architecture has more complexity. Way Prediction requires the use of an auxiliary prediction structure (the way predictor) that must be accurate enough to avoid the performance impact of way mispredictions. It is a speculative technique, and therefore requires verification and recovery hardware, even if predictor accuracy is extremely high. Not only does this additional hardware impact the timing and area of the cache, but it also impacts the complexity of the architecture, as the front-end must be able to stall on a way misprediction. Despite a relatively small loss in performance (around 1%) the serial fetch architecture is still an attractive energy efficient design, which does not have the added complexity or hardware of way prediction and misprediction recovery.

5 Summary

In this paper we have compared the performance of several different front-end architectures. Both our high bandwidth way prediction architecture and our serial fetch architecture successfully combine high bandwidth branch prediction with a scalable and energy efficient instruction cache.

While way prediction can reduce the length of the misprediction pipeline of an architecture, this benefit may not outweigh the architectural costs required to implement way prediction for instruction fetch. The reduction of a single pipeline stage from the front-end of the machine provides an improvement in IPC of only around 1% (2% for perfect way prediction - i.e. no way mispredicts). However, this small improvement carries with it the additional complexity that is required to verify predictions and recover from way mispredictions. The energy benefits of way prediction and the serial fetch architecture are similar. In fact, the way prediction architecture has the potential to expend slightly more energy, as one must consider the energy dissipated by the way predictor and by mispredicted instruction cache data component accesses.

The way predictor architecture includes a number of additional structures and implementation complexity that does not occur in the serial cache architecture. First, there must be extra hardware to detect and recover way mispredictions, including additional control hardware to determine whether a data component access is to obtain an address from the branch prediction engine or from the way misprediction recovery controller. Second, there must be extra hardware to selectively stall results from different data component ports. It may be the case that in a dual ported cache, one port suffered a mispredicted way and the other had a correctly predicted way. Finally, one must consider the extra hardware required to perform the actual way predictions and perform the updates to the way predictor tables.

Our results differ from the prior work [9], where the use of a way prediction cache architecture was

recommended over a serial access cache architecture based on the additional access time for the serial access. However, this additional access time can be pipelined, resulting in what we have found to be a relatively minimal effect on performance. The serial cache is an attractive design for offering simple and energy efficient instruction fetching. Future work will explore tradeoffs between energy efficient data cache techniques. The data cache is not as tolerant of latency (i.e. additional pipeline stages) as the instruction cache, and cache misses can be better tolerated through the out-of-order window.

References

[1] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *27th Annual International Symposium on Computer Architecture*, pages 83–94, Vancouver, Canada, 2000.

[2] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[3] B. Calder and D. Grunwald. Next cache line and set prediction. In *22nd Annual International Symposium on Computer Architecture*, pages 287–296, Santa Margherita Ligure, Italy, 1995.

[4] B. Calder, D. Grunwald, and J. Emer. Predictive sequential associative cache. In *International Symposium on High Performance Computer Architecture*, pages 244–253, San Jose, CA, USA, 1996.

[5] G. Chrysos and J. Emer. Memory dependence prediction using store sets. In *25th Annual International Symposium on Computer Architecture*, pages 142–153, Barcelona, Spain, June 1998.

[6] J. Montanaro et al. A 160 MHz 32b 0.5W CMOS RISC microprocessor. *Digital Technical Journal*, 9(1):49–62, August 1997.

[7] L. Gwennap. Power issues may limit future cpus. *Microprocessor Report*, 10(10), August 1996.

[8] H. Igehy, M. Eldridge, and K. Proudfoot. Prefetching in a texture cache architecture. In *Proceedings of the 1998 Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 133–142, Lisbon, Portugal, 1998.

[9] K. Inoue, T. Ishihara, and K. Murakami. Way-predicting set-associative cache for high performance and low energy consumption. In *International Symposium on Low Power Electronics and Design (ISLPED'99)*, pages 273–275, San Diego, CA, USA, 1999.

[10] R. Kessler. The alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, April 1999.

[11] J. McCormack, R. McNamara, C. Gianos, L. Seiler, N. Jouppi, and K. Correll. Neon: a single-chip 3d workstation graphics accelerator. In *Proceedings of the 1998 EUROGRAPHICS/SIGGRAPH workshop on Graphics Hardware*, pages 123–132, Lisbon, Portugal, 1998.

[12] S. McFarling. Combining branch predictors. Technical Report TN-36, Digital Equipment Corporation, Western Research Lab, June 1993.

[13] M. Powell, A. Agarwal, T. Vijaykumar, B. Falsafi, and K. Roy. Reducing set-associative cache energy via way-prediction and selective direct-mapping. In *34th International Symposium on Microarchitecture*, pages 54–65, Austin, TX, USA, December 2001.

[14] G. Reinman, T. Austin, and B. Calder. A scalable front-end architecture for fast instruction delivery. In *26th Annual International Symposium on Computer Architecture*, pages 234–245, Atlanta, GA, USA, May 1999.

[15] G. Reinman, B. Calder, and T. Austin. Optimizations enabled by a decoupled front-end architecture. *IEEE Transactions on Computers*, 50(4):338–355, April 2001.

[16] G. Reinman, B. Calder, and T. Austin. High performance and energy efficient serial prefetch architecture. In *International Symposium on High Performance Computing*, pages 146–159, Kansai Science City, Japan, 2002.

[17] G. Reinman and N. Jouppi. Cacti version 2.0. http://www.research.digital.com/wrl/people/jouppi/CACTI.html, June 1999.

[18] B. Solomon, A. Mendelson, D. Orenstien, Y. Almog, and R. Ronen. Micro-operation cache: A power aware frontend for variable instruction length ISA. In *International Symposium on Low Power Electronics and Design*, pages 4–9, Huntington Beach, CA, USA, 2001.