In Proceedings of the 4th International Symposium on High Performance Computing (ISHPC), May 2002, (c) Springer-Verlag.

High Performance and Energy Efficient Serial Prefetch Architecture

Glenn Reinman[*] Brad Calder[†] Todd Austin[‡]

[*]Computer Science Department, University of California, Los Angeles
[†]Department of Computer Science and Engineering, University of California, San Diego
[‡]Electrical Engineering and Computer Science Department, University of Michigan

Abstract

Energy efficient architecture research has flourished recently, in an attempt to address packaging and cooling concerns of current microprocessor designs, as well as battery life for mobile computers. Moreover, architects have become increasingly concerned with the complexity of their designs in the face of scalability, verification, and manufacturing concerns.

In this paper, we propose and evaluate a high performance, energy and complexity efficient front-end prefetch architecture. This design, called *Serial Prefetching*, combines a high fetch bandwidth branch prediction and efficient instruction prefetching architecture with a low-energy instruction cache. Serial Prefetching explores the benefit of decoupling the tag component of the cache from the data component. Cache blocks are first verified by the tag component of the cache, and then the accesses are put into a queue to be consumed by the data component of the instruction cache. Energy is saved by only accessing the correct way of the data component specified by the tag lookup in a previous cycle. The tag component does not stall on a I-cache miss, only the data component. The accesses that miss in the tag component are speculatively brought in from lower levels of the memory hierarchy. This in effect performs a prefetch, while the access migrates through the queue to be consumed by the data component.

1 Introduction

At a high-level, a modern high-performance processor is composed of two processing engines: the *front-end processor* and the *execution core*. This producer and consumer relationship between the front-end and execution core creates a fundamental bottleneck in computing, *i.e.,* execution performance is strictly limited by fetch performance.

An energy efficient fetch design that still achieves high performance is 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 [6]. Brooks et al. [2] 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 Alpha 21264. Montanaro et al. [5] found that the instruction cache consumes 27% of power in their StrongARM 110 processor.

The goal of the research presented in this paper is to create a fetch engine that provides:

1. high fetch bandwidth for wide issue processors
2. a complexity effective design that will scale to future processor technologies

3. efficient instruction prefetching to better tolerate memory latencies
4. a design that is as energy efficient as possible, while still achieving the above three goals

In [12, 13], we proposed a fetch architecture that provides high fetch bandwidth for wide issue architectures and can scale to future processor technologies by using a multi-level branch predictor decoupled from the instruction cache. The decoupled branch predictor enabled a highly accurate instruction prefetch architecture.

In this paper, we propose a new prefetch architecture that builds upon our prior design. We examine integrating an energy efficient serial access instruction cache into a high performance instruction prefetch architecture to achieve the above design goals. This design features an intelligent cache replacement and consistency mechanism that reduces the complexity of our prior instruction prefetching scheme from [13].

2 An Energy Efficient Multi-Component Cache

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. We refer to this design as the *parallel* cache. This approach wastes energy in the bitlines and sense amps of the cache as it must drive all associative ways of the data component.

A *serial* cache design breaks up the instruction cache lookup into two components – the tag comparison and the data lookup. The data component is responsible for the majority of the power consumed in the access. If the way is known or predicted before the data component is accessed, we will avoid unnecessarily driving the bitlines of other ways of the cache and decrease the number of necessary sense amps. This design has been used for L2 caches, and recently for data caches on graphics cards [9, 7]. The Alpha 21264 [8] splits the tag and data component of its direct-mapped second level cache, effectively creating a serial L2 cache design. Solomon et al. [17] examined the use of a serial cache along with their Micro-Operation Cache.

The energy efficient cache architecture we use in this paper is the *multi-component cache (MC)*. 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. In this paper, we examine incorporating the MC cache design into our high bandwidth fetch directed prefetching architecture.

3 High Bandwidth Fetch Architecture

In our prior work (shown in Figure 2), we explored an architecture that decoupled the branch prediction architecture from the instruction fetch unit (including the instruction cache) to provide latency tolerance and fetch stream look-ahead. The branch predictor and instruction fetch unit are separated by a queue of fetch addresses (branch predictions) called the *Fetch Target Queue* (FTQ) [14]. 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) [12]. The FTB

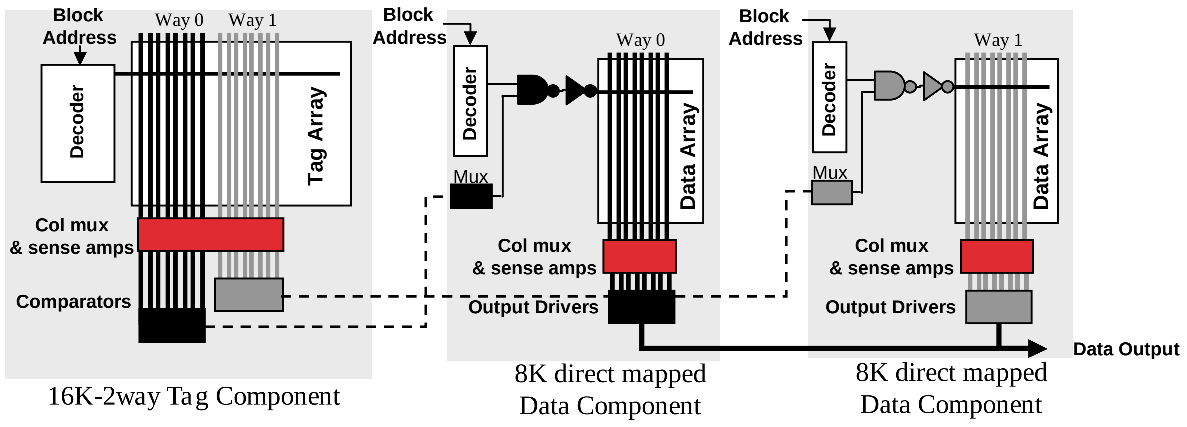


Figure 1: A 16KB 2-way set-associative multi-component (MC) cache. This has the same tag component as the instruction cache, but with multiple data components. Each data component is a direct mapped cache. For a cache of size C that is A-way set associative, there are A direct mapped caches of size$\frac{C}{A}$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

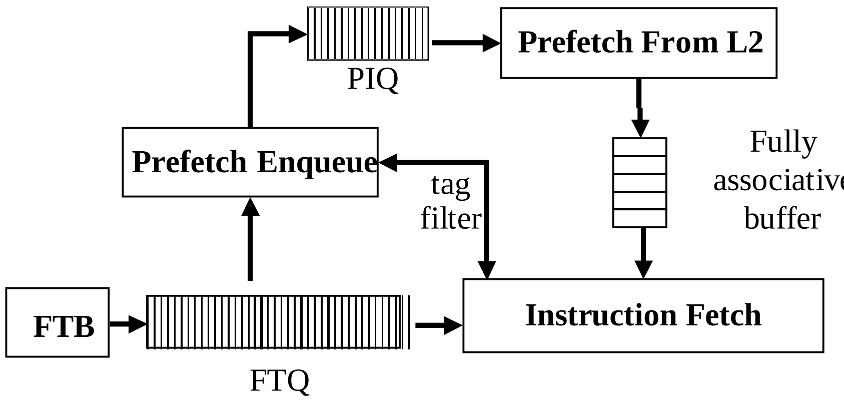


Figure 2: The fetch directed prefetching architecture.

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.

The fetch stream look-ahead provided by the FTQ provides a set of addresses that can be used to perform other optimizations, such as *Fetch-Directed Prefetching* (FDP) [13]. The stream of fetch PCs stored in the FTQ represents the path of execution that the execution core will be following. In that study, we used the stream of fetch PCs in the FTQ to guide instruction cache prefetching.

To increase the accuracy of the prefetches, we examined using a variety of filtering techniques to reduce the number of instruction cache prefetches performed. We used what we call *Cache Probe Filtering*, which uses the instruction cache tag array to verify potential instruction cache prefetch requests. Prefetches are only performed if the cache block is not already in the instruction cache. As it is relatively inexpensive to replicate ports on the instruction cache tag array, a separate port on the tag array can be used to verify FTQ entries for prefetching.

These techniques provide a significant boost to performance, and in this paper, we examine a new architecture that improves the energy efficiency and reduces the complexity of the fetch directed prefetching architecture.

3.1 Complexity Concerns

In Figure 2, the branch predictor feeds fetch blocks into the FTQ where they are consumed by the instruction fetch unit, which contains the instruction cache. Each FTQ entry contains a fetch block PC, a fetch distance, and a branch target. The fetch block prediction stored in a given FTQ entry may span up to five instruction cache blocks.

With FDP, *any* entry in the FTQ can potentially initiate a prefetch. This would require a connection from *each* FTQ entry to the prefetch engine via a multiplexor. This could have substantial design and performance implications. Rather than allowing a prediction to proceed from any entry in the FTQ, we could restrict the number of FTQ entries the architecture is allowed to initiate prefetches from. However, we found that having those restrictions resulted in a significant performance loss. If we restrict prefetching to only the entries at the head of the FTQ, the prefetcher is not able to look far enough ahead to provide a timely prefetch. Restricting prefetching to FTQ entries towards the tail of the FTQ (near the branch predictor) results in reduced coverage, since those FTQ entries are not occupied with fetch blocks a sufficient fraction of the time to provide maximum benefit. Therefore, it is beneficial to have a large window of cache blocks to prefetch from in the FTQ in order to achieve the maximum performance.

For each potential prefetch address, the fetch-directed prefetch architecture in [13] uses a tag port in the instruction cache, to first see if the cache block is in the cache. This is called *Cache Probe Filtering*, and significantly increases the accuracy of the prefetcher. It filters L2 prefetch requests, thereby reducing energy dissipation and saving bus bandwidth. Therefore, the FDP architecture can require a given instance of a cache block to access the instruction cache tag array twice (first for prefetch filtering, and secondly for the actual cache access).

The complexity of processing any FTQ entry to perform a prefetch, and the fact that the FDP architecture requires two tag lookups for every cache block motivated us to examine a design where we effectively move part of the FTQ between the tag and data components of the instruction cache lookup. We call this architecture the *Serial Prefetch* architecture, and describe it in detail in the next section.

4 Serial Prefetch Architecture

To address the issues described in the previous section, we start by decoupling the tag component from the data components of the instruction cache. Figure 3 shows the *Serial Prefetch* (SP) architecture. There are three pipeline stages: branch prediction, tag check, and data lookup. An MC instruction cache is used, and split into tag and data components separated by a queue of cache block requests called the *cache block queue (CBQ)*. The tag component does not block, but the data component blocks on a lookup if the data is not ready, as explained below.

The tag component of the instruction cache consumes fetch block addresses from the FTQ and verifies whether or not the cache blocks within the fetch block are already in the instruction cache. If a cache block is not found by the tag lookup, the prefetch engine can speculatively fetch the block from the L2 cache. The tag component inserts an entry into the CBQ that corresponds to a single cache block request. The data component then consumes this entry and if the block was found to be in the cache (i.e. a known cache hit), it uses the additional state in the CBQ entry to drive the appropriate associative way. We can also use the entries stored in the CBQ to guide cache replacement.

The FTB can provide up to five cache blocks in a single fetch block, but it is expensive (both in terms of power and access time) to add additional ports to the data component of the instruction cache to handle multiple cache blocks. Simply adding extra ports to the tag component of the instruction cache is cheaper than multiporting the data component. This allows the tag component to run ahead of the data component to examine the future cache block request stream of the processor. With the CBQ, the tag component can also run ahead of the data component if the data component has stalled due to a full instruction window or if it is waiting on data from a cache miss.

However, there is a serious consistency issue which must be addressed. If the tag component is allowed to run ahead of the data component, it is possible that a cache data block may be in contention for replacement *after* the tag component has already verified that it is in a particular way of the instruction cache. When the cache block address is sitting in the CBQ waiting to be consumed, its corresponding data block must not be evicted to ensure correctness. To provide for this, we propose the use of a cache consistency mechanism called the *Cache Consistency Table* (CCT), which will be explained in Section 4.3.

We will now examine the structures of the serial prefetch architecture in more detail.

4.1 Prefetch Buffer and Cache Misses

As branch mispredictions do occur, it is not always desirable to bring cache blocks directly into the instruction cache, as they may be on a mispredicted path and could potentially replace a useful block. Therefore, we make use of a separate fully associative structure to hold cache blocks, the *Prefetch Buffer* (PB), which holds cache blocks (analogous to the prefetch buffer from [13]).

The PB works in parallel with the instruction cache for both tag checks and data lookups. The tag components of the instruction cache and PB consume an entry from the FTQ and check each cache block address in the fetch block to determine whether it is a hit in the instruction cache or PB — or if it missed in both and must be retrieved from the lower levels of the memory hierarchy. A new entry is inserted in the CBQ for each cache block that is verified from the FTQ. Then, when the entry is consumed from the CBQ, extra bits contained in the entry will indicate which directed mapped data component (to save energy) should be accessed, and if that component is part of the instruction cache or PB. Note that the PB only has 32 entries (1 KB) and is not drawn to scale with respect to the instruction cache in Figure 3.

If a cache block is not found in the instruction cache or PB during the tag check, then it is prefetched from the lower levels of the memory hierarchy and brought into the PB. When the miss is detected, a PB tag entry is allocated using the consistency mechanism described below. If a PB tag entry cannot be allocated,

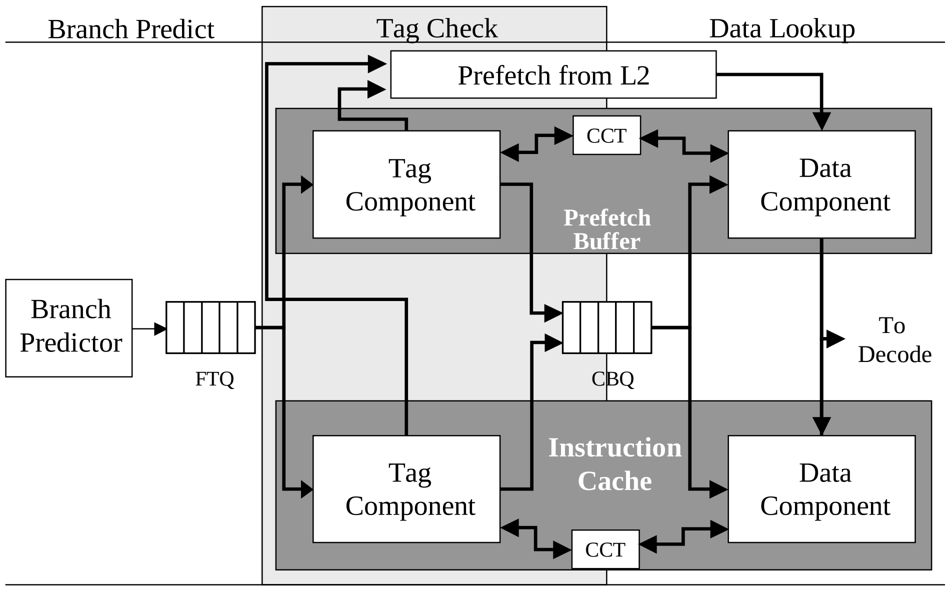


Figure 3: The pipeline for the serial prefetch architecture.

then the tag lookup pipeline stage stalls until an entry in the PB is allocated. The CBQ entry of the block that misses will be marked as cache miss, and it will specify the entry in the PB where the cache block is to be found. When the CBQ entry is consumed by the data lookup, the PB cache block is used and potentially brought into the instruction cache depending upon the cache consistency table, which specifies if a given block can be replaced or not.

This approach does not allocate blocks into the instruction cache until they are used by the data component. This reduces cache pollution caused by branch mispredictions. We found this to perform better than allocating the cache block during the tag component pipeline stage when the initial miss occurs.

4.2 Cache Block Queue

The CBQ holds a cache block address, *block location* bits, *instruction cache way* bits, and *PB index* bits. The block location bits represent whether the cache block that the entry represents is in the instruction cache, in the PB, or is to be brought into the PB from another level of the memory hierarchy. The way bits are fed into the data component of the instruction cache on an instruction cache hit. These bits indicate which direct mapped component should be activated (i.e. what data way the tag component found the data in – the output from the comparators of the tag array). When the cache block hits in the PB, the PB index is used to keep track of the location of the cache block in the PB. If the cache block is prefetched from lower levels of the memory hierarchy, the PB index holds the location where the prefetched cache block will be stored.

The size of the CBQ can be used to control the amount of prefetching that occurs. The larger the queue, the more the tag comparator can be allowed to run ahead of the data output components, and the more cache blocks (not found in the instruction cache by the tag component) that can potentially be brought into the PB. The further ahead the tag component runs of the data components, the earlier the prefetch can occur and the more memory latency that can be hidden. However, there is also a greater chance of speculating down a mispredicted control path. Therefore the size of the CBQ trades the benefit obtainable from prefetching with the amount of power potentially wasted on mispredicted control paths (similar to the tradeoffs inherent in FTQ size). The CBQ is flushed on a branch misprediction.

4.3 Consistency Mechanism

Because the tag component can verify cache blocks far in advance of the data component and we perform replacement in the data lookup pipeline stage, we need some consistency mechanism to guarantee that cache blocks verified by the tag component are not evicted during cache replacements before they can be accessed in the data lookup stage. We maintain an extra table, called the *cache consistency table (CCT)*, to guarantee this. The CCT is a tagless buffer, with one entry for every cache block in the instruction cache, but with much smaller blocks – each CCT block only holds an N-bit counter. For example, assuming the use of a 3-bit counter, a 16KB 2-way associative instruction cache would only need a 320 byte table (a structure roughly 1% of the size of the instruction cache).

The counter stored in each CCT entry represents the *number of outstanding verified cache block requests* sitting in the CBQ for the corresponding data block in the instruction cache. When the tag component verifies a cache block in the tag check stage, the N-bit counter in the CCT corresponding to that cache block is incremented. When a data component accesses an instruction cache block, the N-bit counter in the CCT corresponding to that cache block is decremented. On a misprediction or misfetch, the CCT is flushed (set to zero), just as the CBQ is also flushed. The mapping from instruction cache to CCT is implicit and does not require tags – since both structures have the same number of entries and associativity, they have identical decoders.

This consistency mechanism also extends to the prefetch buffer. The PB has its own dedicated CCT as shown in Figure 3. If the desired cache block is not found in the instruction cache, but is in the PB, a N-bit counter associated with this block in the PB is incremented each time that block is placed in the CBQ. This is necessary to preserve cache consistency so that a cache block in the PB that a CBQ entry points to is not replaced.

The CBQ provides a means to look ahead at the behavior of the instruction cache. In addition to reducing energy by only accessing the data component known to contain the desired cache block, the CBQ can also *help guide instruction cache or PB replacement*, with the help of the CCT. When a cache block is brought into the instruction cache from the PB, a block is chosen for replacement from the set that has a zero CCT entry. This ensures that a cache block, which a later CBQ entry wants to use, does not get removed from the cache. This policy overrides the standard LRU replacement policy of the instruction cache. If all cache blocks in a particular cache set are marked in the CCT (meaning no replacement for the new cache block exists), then the block is not put into the instruction cache, and instead just stays in the PB until used again or is replaced. Similarly, the replacement policy ensures that entries in the PB with non-zero N-bit counters are not replaced by new prefetches until they are have been using by the CBQ entry that incremented their N-bit counter. In this manner, the PB acts as a flexible depository of instruction cache blocks for contended cache sets – directed by the CBQ and CCT.

On a branch misprediction, all entries in the PB have their CCT entries cleared (i.e. set to zero) – but the PB is *not* flushed. New PB entries are allocated based on a LRU replacement policy of the entries that have cleared N-bit counters. If an entry in the PB with a cleared N-bit counter matches a desired cache block in the tag check stage, the N-bit counter is incremented, and any second level cache access is avoided. This way, when a mispredicted short forward jump or other wrong-path prefetch is encountered, the prefetching performed on the mispredicted path can be reused. When the processor encounters a miss in both the PB and instruction cache, and there are no entries in the PB with cleared N-bit counters, the tag components of both the PB and instruction cache stall until the N-bit counter of a PB entry has been decremented to provide a space for the desired cache block. This will happen either once the CBQ drains far enough to clear an N-bit counter or if a branch misprediction is detected.

In this paper, we use 5-bit CCT counters when using a 32 entry CBQ, and 3-bit CCT counters when using a 12 entry CBQ.

5 Methodology

The simulator used in this study was derived from the SimpleScalar/Alpha 3.0 tool set [3], 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 5 of the SPEC95 C benchmarks plus 2 of the SPEC2000 C benchmarks. We only selected benchmarks that exhibited adequate instruction cache miss behavior in presenting our results – but there was no performance degradation for benchmarks without instruction cache pressure. The programs were compiled on a DEC Alpha AXP-21164 processor using the DEC C and C++ compilers under OSF/1 V4.0 operating system using full compiler optimization (`-O4 -ifo`). Each benchmark was fast forwarded before actual simulation as described by Sherwood et. al. [16]. Their approach uses basic block fingerprinting to determine how far to fast forward in order to have accurate simulation points.

5.1 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 [4]). Therefore, a load only waits on a store it is really 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.

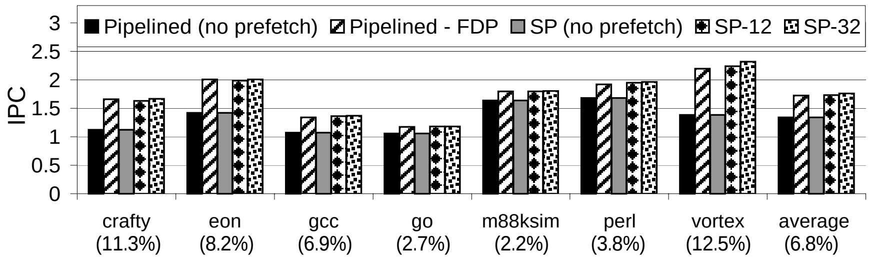
There is an 8 cycle minimum branch misprediction penalty. The processor has 8 integer ALU units, 4-load/store units, 2-FP adders, 2-integer MULT/DIV, and 2-FP MULT/DIV. The latencies are: Int ALU 1 cycle, Int MULT 7 cycles, Int DIV 12 cycles, FP ALU 4 cycles, FP MULT 4 cycles, and FP DIV 12 cycles. All 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 [10], 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.

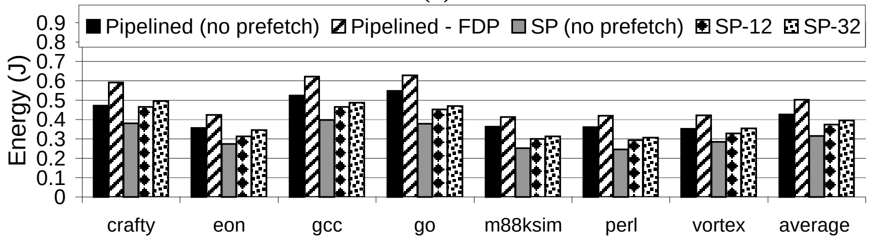
5.2 Memory Hierarchy

We rewrote the memory hierarchy in SimpleScalar to model bus occupancy, bandwidth, and pipelining of the second level cache and main memory. We provide results for instruction caches with a single ported data component and dual ported tag component. We found that this port configuration provides the best tradeoff between performance and energy dissipation. Additional tag or data ports impacted energy dissipation more than they helped performance. The data cache for each configuration is a 4-way set associative 32KB cache with 32 byte lines. Results are gathered using a 16KB 2-way associative instruction cache.

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.



(a)



(b)

Figure 4: IPC and Energy results

5.3 Energy Model

The amount of energy consumed by a circuit influences layout issues, power-supply requirements, thermal considerations, and even reliability [11]. We are interested in building an architecture that combines high performance with energy efficiency.

The energy data we need to generate results is gathered using the new CACTI cache model version 2.0 developed by Reinman and Jouppi [15]. 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. Since we are concerned with instruction caches, we only examine cache writes as replacements from lower levels of the memory hierarchy. Also, we modified CACTI 2.0 to support extra ports on just the tag array of the cache.

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 front-end requests, not from data cache misses), and other auxiliary structures that have been introduced (CBQ, CCT, and PB). For each, we modified the BITOUT, ADDRESS BITS, and block size parameters appropriately. When we report energy dissipation results in Joules, this includes the power dissipated by all the above listed front-end structures.

6 Results

Figures 4(a) and (b) present IPC and Energy results for five architectures. The first two bars represent the pipelined parallel instruction cache from prior work – with and without FDP using cache probe filtering: Pipelined (no prefetch) and Pipelined-FDP. The remaining three bars represent the serial prefetch architec-

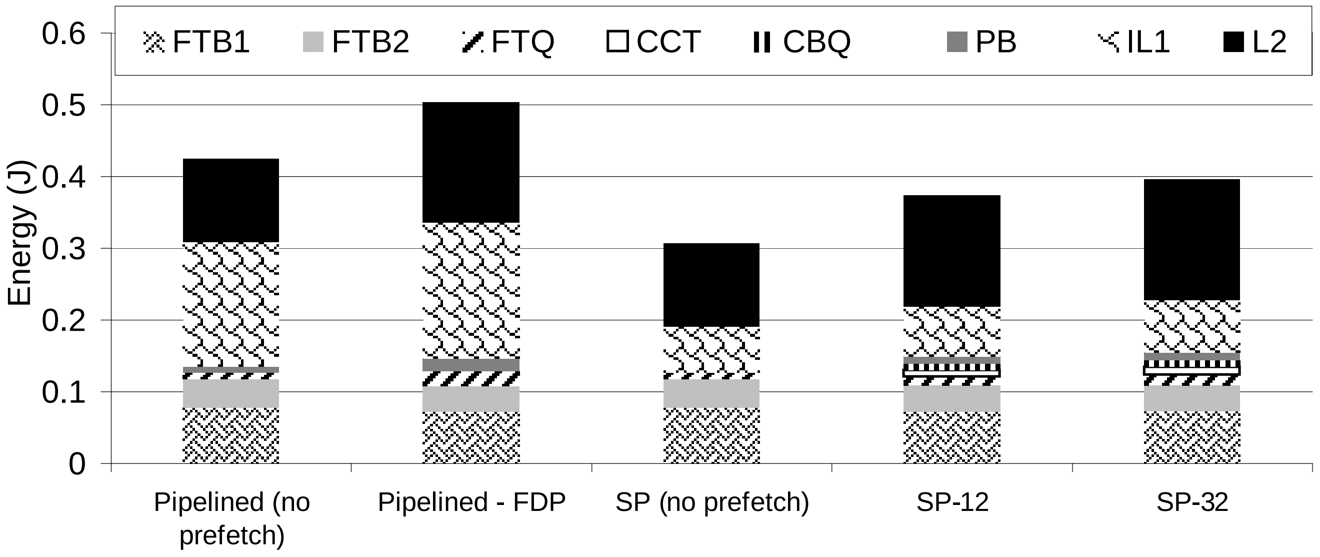


Figure 5: Energy breakdown for 16KB 2-way set associative cache.

ture with the MC cache. SP (no prefetch) is the serial prefetch architecture with a 0 entry CBQ. The other two bars represent the serial prefetch architecture with a 12-entry CBQ (SP-12) and a 32-entry CBQ (SP-32). All instruction caches are 16KB 2-way set associative, with two tag array ports and a single data array port. Instruction cache miss rates are shown under each benchmark in (a).

Because both the pipelined parallel cache and the serial prefetch architecture feature instruction cache accesses across 2 pipeline stages, the performance of these architectures without any form of prefetching (i.e. without a CBQ) is identical. However, the use of the MC cache in the serial prefetch architecture provides a 24% reduction in energy dissipation on average over the pipelined parallel cache. The use of a 32-entry CBQ provides a 31% improvement in IPC over the architecture with a 0-entry CBQ (no prefetch).

For all benchmarks, the serial prefetch architecture with a 12 or 32 entry CBQ is able to perform as well or better than the pipelined parallel cache with fetch directed prefetching. The SP-32 architecture is able to use 21% less energy on average than the parallel pipelined cache with FDP. The SP-12 architecture is able to use 26% less energy.

The difference between the SP-12 and SP-32 architectures is relatively small for most benchmarks, with the exception of `vortex`. This benchmark is plagued by a larger instruction cache miss rate than other benchmarks and can therefore take full advantage of the intelligent replacement policy provided by the CCT in the serial prefetch architecture.

Finally, we have traded complexity in the design of both the FTQ and the FDP prefetch mechanism for the addition of the CBQ and CCT to the Serial Prefetch architecture. By consolidating prefetch verification and enqueue to a single site (the tag component of the MC cache), we have reduced the amount of wire needed around the FTQ. This could have a large impact at future technology sizes. Moreover, we have reduced the number of accesses to the tag array of the instruction cache and the number of times that an FTQ entry needs to access an address generator. The CBQ, CCT, and PB are all small structures which will scale well to future technology sizes [1]. Also, by integrating prefetching into the regular operation of the instruction cache, we have eliminated some of the scheduling complications that might arise from having three distinct sources of memory requests (now the instruction cache demand misses and instruction prefetching requests originate from the same source).

Figure 4(b) showed that the Pipelined cache architectures dissipate considerably more energy than the

Serial Prefetch architectures. Figure 5 explores this more closely by breaking down the average energy consumption of the 7 benchmarks by front-end component. The instruction cache, L2 cache, and first level FTB dissipate the most energy of the front-end structures shown in this Figure. The FTQ, CBQ, and CCT dissipate very little energy relative to these structures. Prefetching adds to the energy dissipation of the front-end – but, the SP-32 technique in Figure 4(b) dissipate slightly less energy on average than the Pipelined (no prefetch) architecture, which does *not* even include any form of prefetching. This is due to the energy efficient MC cache design.

7 Summary

In this paper we examined integrating an energy efficient instruction cache into a high fetch bandwidth architecture with prefetching. The goal was to not sacrifice any performance, but at the same time to reduce the energy footprint and the complexity of the high fetch bandwidth architectures.

We examined the complexity inherent in the design of a fetch-directed prefetching architecture, and proposed a novel serial prefetch architecture that decouples the tag and data components of the instruction cache. This architecture reduces the complexity of the FTQ and energy dissipation over our prior fetch-directed prefetching architecture. This is made possible by use of a new cache consistency mechanism (called the CCT) that coordinates the tag and data components of the instruction cache and provides an improved form of cache replacement.

The best performing serial prefetch architecture provides a 31% improvement in IPC over the parallel cache with no form of prefetching, and can reduce energy dissipation by 21% over a parallel instruction cache with fetch directed prefetching. This design is able to scale well to more highly associative caches. The intelligent replacement mechanism can even provide high performance in applications with severe instruction cache thrashing. Finally, the serial prefetch architecture exhibits less complexity along the critical timing path by reducing wire congestion around the FTQ, and only requiring one tag lookup for a cache block when it is prefetched.

Acknowledgments

We would like to thank Keith Farkas, Norm Jouppi, and the anonymous reviewers for providing useful feedback on this paper. This work was funded in part by Semiconductor Research Corporation contract number 2001-HJ-897, by DARPA/ITO under contract number DABT63-98-C-0045, and a grant from Intel Corporation.

References

[1] V. Agarwal, M. Hrishikesh, S. Keckler, and D. Burger. Clock rate versus ipc: The end of the road for conventional microarchitectures. In *27th Annual International Symposium on Computer Architecture*, 2000.

[2] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *27th Annual International Symposium on Computer Architecture*, 2000.

[3] 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.

[4] G. Chrysos and J. Emer. Memory dependence prediction using store sets. In *25th Annual International Symposium on Computer Architecture*, June 1998.

[5] J. Montanaro et al. A 160 mhz 32b 0.5w cmos risc microprocessor. In *Digital Technical Journal*, August 1997.

[6] L. Gwennap. Power issues may limit future cpus. Microprocessor Report, August 1996.

[7] H. Igehy, M. Eldridge, and K. Proudfoot. Prefetching in a texture cache architecture. In *Proceedings of the 1998 Eurographics/SIGGRAPH Workshop on Graphics Hardware*, 1999.

[8] R. Kessler. The alpha 21264 microprocessor. In *IEEE Micro*, April 1999.

[9] 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*, 1999.

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

[11] J. Rabaey. *Digital Integrated Circuits*. Prentice Hall Electronics and VLSI Series., 1996.

[12] G. Reinman, T. Austin, and B. Calder. A scalable front-end architecture for fast instruction delivery. In *26th Annual International Symposium on Computer Architecture*, May 1999.

[13] G. Reinman, B. Calder, and T. Austin. Fetch directed instruction prefetching. In *32st International Symposium on Microarchitecture*, November 1999.

[14] G. Reinman, B. Calder, and T. Austin. Optimizations enabled by a decoupled front-end architecture. In *IEEE Transactions on Computers*, April 2001.

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

[16] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[17] 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*, 2001.