Evaluating the Energy Efficiency of Trace Caches

University of Virginia, Dept. of Computer Science Tech. Report CS-2003-19

Michele Co and Kevin Skadron
Department of Computer Science
University of Virginia
{micheleco, skadron}@cs.virginia.edu

Abstract

Sequential trace caches are highly energy and power-efficient. Fetch engines which include a sequential trace cache provide higher performance for approximately equal area at a significant energy and power savings. The results of our preliminary experiments show that sequential trace caches are a power-efficient design.

Previous work has evaluated the trace cache design space with respect to performance. In addition, some previous work has evaluated power-efficiency techniques for trace caches. This work evaluates the trace cache design space considering not only performance but also energy and power. In addition, we compare fetch engine designs which include trace caches with fetch engine designs have instruction caches only.

We perform a set of fetch engine area and associativity experiments as well as a next trace predictor design space exploration. We find that when examining performance and average fetch power, fetch engines with trace caches may not seem appealing, but when examining energy-delay and energy-delay-squared, the benefits of a trace cache become clear. Even if average fetch power is increased due to the increased fetch engine area, the energy-efficiency is still improves with a trace cache due to faster execution and more opportunities for clock gating, making the trace cache superior in terms of energy-delay and energy-delay-squared products.

Results of current experiments show that sequential trace cache designs compare very favorably to instruction-cache-only designs with respect to power and energy consumption. Our preliminary results show that overall sequential trace caches clearly outperform instruction-cache-only designs with better energy-efficiency. In examining the best design points of the fetch engines examined, a 343KB, 4-way set associative trace cache fetch engine outperforms a 292KB instruction-cache-only fetch engine by 5% for integer benchmarks and 1% for floating point benchmarks. In addition, it does so using 68.3% less average fetch power, 70.3% less energy, 67.7% less energy-delay, and 69.1% less energy-delay-squared than a 292KB, 2-way set associative instruction-cache-only fetch engine.

Our exploration of the Jacobson next trace predictor design space indicates that a hybrid next trace predictor consisting of a 1KB correlating table, 8KB secondary table, and 4-entry return history stack would be the best choice in terms of power, performance, and energy.

1 Introduction

Newer processor technologies enable higher transistor density on a chip, therefore power consumption is an important consideration for new designs. In addition, today's desktop CPU's become tomorrow's laptop CPU's so evaluating the energy profile of microarchitectural designs is also important.

Research shows that the fetch unit contributes a large portion of total power consumption in a microprocessor [16]. Therefore, evaluations for any microarchitectural design for the fetch unit should consider power and energy in addition to performance.

High performance superscalar processor organizations can be divided into an instruction fetch mechanism and an instruction execution mechanism. The instruction fetch mechanism is responsible for producing instructions for the execution mechanism to consume. Because of this producer-consumer relationship, instruction fetch bandwidth is a bottleneck to high performance.

The fetch unit's role is to feed the dynamic instruction stream to the execution unit. However, instruction caches store instructions in static program order. Due to the presence of taken control flow instructions, some of the instructions fetched from the instruction cache are useless.

Several high fetch bandwidth mechanisms such as branch address cache [26], subgraph predictor [7], collapsing buffer [6], multiple-block ahead predictor [22], and trace cache [14, 18, 19, 21] have been proposed. Each of these high fetch bandwidth mechanisms have drawbacks in terms of complexity. They require additional levels of indirection, moderate to highly

interleaved instruction caches, or complex alignment networks. In addition, some of these mechanisms are on the critical path. Many of these mechanisms have not been evaluated in terms of power or energy.

Most trace cache implementations do not suffer from the problems of additional levels of indirection or the need for interleaving or complex alignment networks. The trace cache is a special type of instruction cache, which stores instructions in dynamic program order. Traces are generally built at instruction commit time, which puts trace-building off the critical path. Trace caches help address the problem of fetching unused instructions in cache lines and has the potential to improve power-efficiency over the traditional instruction cache. Our work uses the trace cache described by Rotenberg *et al.* [19, 21] as a starting point and is the main focus of this work.

Some research [9, 10] has explored ways to reduce the power consumption of trace caches. Hu *et al.* [10] showed that sequentially accessing the trace cache and instruction cache has a significant power savings over accessing the two structures simultaneously. In addition, some research has worked towards improving traditional instruction cache energy consumption without adversely affecting processor performance or on-chip energy consumption [1, 15, 27]. However, little research has examined the relative power-energy-performance tradeoff between fetch organizations which have only instruction caches and fetch organizations which have a combination of instruction cache and trace cache. This paper examines the power-energy-performance tradeoffs of these two broad categories of fetch engine organization.

The rest of the paper is organized as follows. In Section 2, we present related work. We present our experimental models in Sections 3 and 4. Experimental results are presented in Section 5. Conclusions and future work are presented in Section 6.

2 Related Work

Because instruction fetch bandwidth is a performance bottleneck, researchers search for techniques to improve the quantity and/or quality of fetched instructions. The trace cache is one such mechanism. Trace caches store dynamic sequences of instructions, therefore potentially increasing the number of useful instructions fetched. Friendly, Patel, and Patt [8] and Rotenberg, Bennett, and Smith [19, 21, 20] performed comprehensive studies of the trace cache design space with respect to performance. We seek to perform a similar design space study to evaluate power, energy, and performance tradeoffs on a more current processor pipeline.

Hu *et al.* [10] evaluated the power efficiency of conventional trace caches as compared to sequential trace caches. They found that probing the trace cache sequentially derived significant power savings over parallel probing. Our work uses this finding as a starting point for our design space exploration.

Hu *et al.* [9] also compared the conventional trace cache (CTC), sequential trace cache (STC), and dynamic direction prediction based trace cache (DPTC) for power efficiency and performance. CTC is a trace cache which is probed in parallel with the instruction cache. STC is a trace cache in which the trace cache is probed first, and then only on a trace cache miss is the instruction cache probed. DPTC improves upon STC by collecting history information to dynamically select whether to probe the trace cache or the instruction cache. They found that DPTC exhibits less performance loss than the sequential trace cache at similar power consumption. Our work seeks to evaluate more of the the trace cache design space focusing on performance, power and energy and also compares the sequential trace cache fetch unit to an instruction-cache-only fetch unit.

3 Simulation Techniques

All experiments in this work use the HotSpot [24] infrastructure because it updates Wattch with a power model based on the Alpha 21364. HotSpot is a cycle accurate, power and thermal functional simulator based on Wattch [4] and Simplescalar [5], although we turn off the thermal modeling for this work. The base HotSpot simulator (which supports only an instruction-cache fetch model) was extended to include a conventional trace cache fetch mechanism. A next trace predictor (NTP) [13] is used to make predictions about future traces to execute. The following subsections describe the simulation methodology in more detail.

3.1 Baseline HotSpot Power-Performance Simulator

The HotSpot power model is based on power data for the Alpha 21364 [2]. The 21364 consists of a processor core identical to the 21264, with a large L2 cache and (not modeled) glueless multiprocessor logic added around the periphery. Wattch version 1.02 [4] is used to provide a framework for integrating power data with the underlying SimpleScalar [5] architectural model. Power data was for 1.6 V at 1 GHz in a 0.18μ process, so Wattch's linear scaling was used to obtain power for 0.13μ, V_{dd}=1.3V, and a clock speed of 3 GHz. This is the same power model used in the HotSpot work [24].

In HotSpot, the SimpleScalar *sim-outorder* microarchitecture model was augmented to model an Alpha 21364 as closely as possible. The microarchitecture and corresponding Wattch power interface were extended; extending the pipeline and breaking the centralized RUU into four-wide integer and two-wide floating-point issue queues, 80-entry integer and floating-point merged physical/architectural register file, and 80-entry active list. First-level caches are 64 KB, 2-way, write-back, with 64B lines and a 2-cycle latency; the second-level is 4 MB, 8-way, with 128B lines and a 12-cycle latency; and main memory has a 225-cycle latency. The branch predictor is similar to the 21364's hybrid predictor, and the instruction-cache-only performance simulator is improved by updating the fetch model to count only one access (of fetch-width granularity)

per cycle. The only features of the 21364 that are not modeled are the register-cluster aspect of the integer box, way prediction in the I-cache, and speculative load-use issue with replay traps. The microarchitecture model is summarized in Table 1.

Processor Core	
Active List	80 entries
Physical registers	80
LSQ	64 entries
Issue width	6 instructions per cycle (4 Int, 2 FP)
Functional Units	4 IntALU,1 IntMult/Div, 2 FPALU,1 FPMult/Div, 2 mem ports
Memory Hierarchy	
L1 D-cache Size	64 KB, 2-way LRU, 64 B blocks, writeback
L1 I-cache Size	64 KB, 2-way LRU, 64 B blocks both 2-cycle latency
L2	Unified, 4 MB, 8-way LRU, 128B blocks, 12-cycle latency, writeback
Memory	225 cycles (75ns)
TLB Size	128-entry, fully assoc., 30-cycle miss penalty
Branch Predictor	
Branch predictor	Hybrid PAg/GAg with GAg chooser
Branch target buffer	2 K-entry, 2-way
Return-address-stack	32-entry

Table 1: HotSpot's simulated processor microarchitecture.

In order to more closely study the efficiency of the fetch engines, we chose a highly parallelizing execution core. We altered the HotSpot simulated microarchitecture to have 128 fetch queue entries, 128 register rename entries, and 128 load/store queue entries. In addition, as many as 16 instructions can be issued, executed, and committed in one cycle. Thus, a maximum of 16 IPC is possible with a perfect fetch engine and perfectly parallel code.

4 Trace Cache Model

4.1 Trace Cache

The trace cache that is modeled in the experiments is the one described by Rotenberg [20]. The trace cache consists of a Jacobson path-based next trace prediction mechanism [13] which predicts the next trace to be fetched, outstanding trace buffers to hold in-flight predicted traces, and the trace cache itself. The trace cache can be configured to be probed in parallel or in sequence with the instruction cache. For the purposes of this work, we have configured the trace cache to be probed first as in the sequential trace caches (STC) in Hu's work [10]. We only probe the instruction cache on a trace cache miss. Therefore, on a trace cache miss, there is a one cycle delay before the instruction cache may be probed. Figure 1 shows our trace cache model. In Wattch, the trace cache's power was modelled as an array structure, similar to an instruction cache, with one read and one write port.

4.2 Next Trace Prediction

Different research defines traces in various ways. Our work uses the definitions used in the work of Jacobson *et al.* [13] since we use their path-based next trace predictor in our experiments. In that work, a trace has a maximum of 16 instructions, as many as 7 branches (6 internal branches, plus a possible 7th terminating branch). Indirect branches (jump via a register's contents, a return instruction, a system call instruction, etc.) terminate a trace.

The next trace predictor (NTP) [13] uses path history information (recently committed traces) to make predictions much like a GAs or global history branch predictor. It uses the same hashing algorithm as is used for the inter-task prediction index generation mechanism for multiscalar processors [12] with *Depth, Older, Last, Current* set to: 7,3,6,8. *Depth* represents the number of hashed trace identifiers to use in the hashing algorithm not including the most recently predicted hashed trace identifier (referred to as the current trace identifier). *Current* and *Last* specify the number of bits to use in the hash from the most recent hashed trace identifier and next most recent hashed trace identifier, respectively. *Older* specifies the number of bits to use for the remaining *Depth - 1* hashed trace identifiers. This information is combined with trace history to index a table that makes a prediction about the next trace to be fetched. In our configuration, 8 previous trace identifiers are hashed

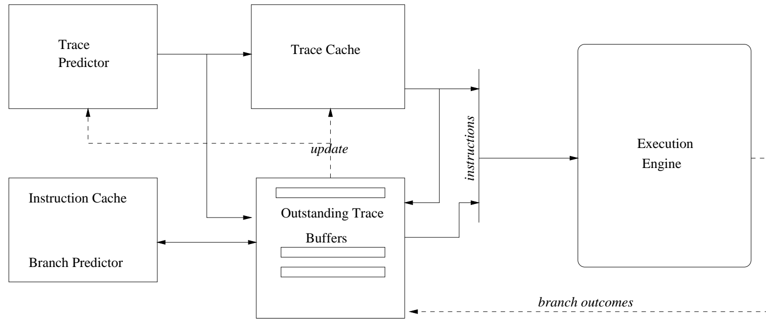


Figure 1: Trace cache model (Patterned after trace cache model figure in Rotenberg [20]).

together to get indexes into the 64k-entry correlating (primary) table, and into the 32k-entry secondary table. Each table contains a predicted trace identifier, and a saturation counter (2 bits for the primary table, 4 bits for the secondary table) that indicates how accurately the table entry is predicting. A selector mechanism chooses the more accurate table.

To further increase the accuracy of the next trace predictor, a 128-entry return history stack (RHS) is used. The current trace history is pushed onto the stack for every call instruction executed, and popped for every return instruction execution. Only the top of stack pointer is recovered on a misprediction.

In the event of a misprediction, no alternate trace prediction is generated. Instead, the traditional instruction cache and branch predictor are used to make a new guess about the remainder of the trace. Hu *et al.* [10] showed that serial accessing of the trace cache and instruction cache can provide a significant fetch power reduction over parallel accessing of the trace cache and instruction cache.

The Wattch power model was augmented to model the Jacobson hybrid next trace predictor. The correlating table, secondary table, return history stack and path history register are each modeled as array structures, similar to the branch predictor structures, with one read and one write port.

4.3 Outstanding Trace Buffer

The outstanding trace buffer (OTB) is a mechanism used to hold in-flight traces (thus, only traces predicted by the next trace predictor). When an entire trace commits, the trace is written to the trace cache (if needed) and the OTB entry is recovered. All experiments use 128 OTB entries. OTB entries also contain information about how to recover from a mispredicted trace.

The Wattch power model was augmented to model the outstanding trace buffer as an array structure with two read ports and one write port. It is modelled similar to a register file. The fetch and mispredict recovery mechanisms share one read port and commit time has a devoted read port. The single write port is shared between fetch and mispredict recovery mechanisms.

4.4 Experimental Trace Cache Configurations

In our experiments which contain a trace cache, there is an instruction cache which serves as backup in the case of a trace cache miss. The backing instruction cache we chose has 512 sets, 64 byte lines, 2-way set associativity, and uses an LRU replacement algorithm. The parameters that were held constant are shown in Table 2.

Component	Configuration
I-cache	512 set, 64B line, 2-way set associative, LRU
Branch predictor (including BTB)	Hybrid: 4K-entry PAg, 4K-entry GAg (12-bit history register), 4k-entry GAg chooser 2k-entry, 2-way set associative BTB 32-entry RAS
OTB	128 entries
NTP	64K-entry correlating table, 32K-entry secondary table, 128-entry RHS

Table 2: Parameters held constant for experiments.

The trace cache itself has varying associativities (1-, 2-, and 4-way) but the replacement policy is fixed to LRU, and the line size is fixed at 1 trace (maximum of 16 instructions and 6 branches). Table 3 shows the fetch engine areas used in each experiment. The area of the total fetch engine area used for the trace cache itself is listed alongside each fetch engine area. The remaining fetch engine area is calculated by totalling the area of the instruction cache, branch predictor, outstanding trace buffer, and next trace predictor for the trace cache experiments. For the trace cache experiments, the area of the instruction cache, branch predictor (including BTB), outstanding trace buffer, and next trace predictor are held constant (See Table 2).

Fetch Engine Area	Trace Cache Area
231 KB	16 KB
247 KB	32 KB
279 KB	64 KB
343 KB	128 KB
471 KB	256 KB
727 KB	512 KB

Fetch Engine Area	Instruction Cache Area
100 KB	32 KB
164 KB	64 KB
292 KB	128 KB
548 KB	256 KB
1060 KB	512 KB

Table 3: Fetch engine area and corresponding trace cache and instruction cache areas used in experiments. Cache area is part of the fetch engine area total.

4.5 Experimental Instruction Cache Configurations

When a configuration uses only an instruction cache, we use a 2-way set associative cache with 64 byte lines, and LRU replacement. The fetch engine areas and the area of the total fetch engine area used by the instruction cache are listed in Table 3. For the instruction-cache-only experiments, the fetch engine area is comprised of the area for the instruction cache and the branch predictor, with the branch predictor area held constant as described in Table 2. Table 3 highlights the instruction cache sizes used for different configurations.

4.6 Benchmarks

We evaluate our results using benchmarks from the SPEC CPU2000 suite. The benchmarks are compiled and statically linked for the Alpha instruction set using the Compaq Alpha compiler with SPEC *peak* settings and include all linked libraries but no operating-system or multiprogrammed behavior. For each program, we fast-forward to a single representative sample of 500 million instructions. The location of this sample is chosen using the data provided by Sherwood *et al.* [23]. Simulation is conducted using SimpleScalar's EIO traces to ensure reproducible results for each benchmark across multiple simulations. Twelve of the SPEC2000 benchmarks were used. Seven of the integer benchmarks (gzip, gcc, crafty, parser, eon, perlbmk, and vortex) and 5 of the floating point benchmarks (wupwise, mesa, art, facerec, and ammp) were chosen.

After loading the SimpleScalar EIO checkpoint at the start of our desired sample, it is necessary to warm up the state of large structures like caches and branch predictors. When we start simulations, we first run the simulations in full-detail cycle-accurate mode (but without statistics-gathering) for 100 million cycles to train the caches—including the L2 cache—and the branch predictor. This interval was found to be approximately sufficient using the MRRL technique proposed by Haskins and Skadron [11].

5 Experimental Results

5.1 Fetch Engine Area Exploration

A fetch engine area comparison of a sequential trace cache using the next trace prediction mechanism of Jacobson et al. [13] and traditional instruction-cache-only fetch unit is performed using the areas listed in Table 3. Associativity and fetch engine area were varied and the IPC, average fetch power, fetch energy, energy-delay, and energy-delay-squared were analyzed. Since current CPU designs are increasingly using conditional clocking techniques to reduce power consumption when hardware is not in use [4], we calculate the power and energy metrics using Wattch's conditional clocking method which scales power linearly with port or unit usage: when the port or unit is not in use, 10% of its maximum power dissipation is charged as opposed to zero power as a way to account for leakage energy. Since it was difficult to achieve equal areas for each fetch engine, in our analysis, we compare the best trace cache area to the instruction cache area which is at the point of diminishing returns. The results plotted in the graphs are the averages of the benchmarks that were run. In our power and energy results, the instruction-cache-only results have a linear slope which changes with fetch engine area, while the trace cache results have a sublinear slope. This is due to clock gating. If conditional clocking were not considered, the trace cache results would also have a slope with fetch engine area similar to what is observed for instruction-cache-only fetch engines.

5.1.1 Analysis of Results for Integer Benchmarks

We find that for approximately equal fetch engine area, a sequential trace cache plus instruction cache design clearly wins when considering energy efficiency and performance together. This is shown in Figures 2–6. For the integer benchmarks, all of the trace cache configurations of approximately equal area clearly outperform the instruction-cache-only configurations in performance, average fetch power, fetch energy, energy-delay and energy-delay-squared. In fact, the graphs show that any fetch engine which contains a trace cache clearly outperforms all instruction-cache-only fetch engines with better energy efficiency.

As an example, it is clear that the maximum gain in performance (IPC) can be gained by the 343 KB, 4-way set associative trace cache fetch engine configuration. It is also clear that the maximum performance gain for the instruction-cache-only configurations is the 292 KB instruction-cache-only fetch engine area. When compared to the 292 KB instruction-cache-only design, a 343 KB, 4-way set associative trace cache configuration shows a 5.1% performance improvement, 68.3% reduction in average fetch power, and a 70.3% reduction in fetch energy. The significant improvement in the average fetch power of the trace cache configuration over the instruction-cache-only fetch engine is due to clock gating. Basically, the fetch engine which contains the trace cache is used in a more power-efficient way than the instruction-cache-only fetch engine. In addition, the energy-delay is 67.7% lower and the energy-delay-squared is 69.6% lower than the 292 KB instruction-cache-only fetch engine.

5.1.2 Analysis of Results for Floating Point Benchmarks

For the floating point benchmarks, the trend is essentially the same. Overall, the fetch engine configurations which contain a trace cache clearly outperform the instruction-cache-only fetch engines. In addition, the fetch engine configurations which contain trace caches clearly show better energy efficiency than instruction-cache-only configurations (See Figures 2-6). One difference between the results for the floating point benchmarks compared to the integer benchmarks is that for the floating point benchmarks, changes in the instruction-cache-only area have no effect on performance because floating point benchmarks have a small text size, often contain looping code, and are reasonably predictable.

Again, as an example, we compare the trace cache configuration which gains maximum performance against the best instruction-cache-only configuration. The maximum performance gain for trace cache configurations is seen at 279 KB with 4-way set associativity but is less significant than the performance gain for the integer benchmarks. The 279 KB, 4-way set associative trace cache configuration shows a 1% performance improvement over a 164 KB instruction-cache-only fetch engine configuration. The 279 KB, 4-way set associative trace cache fetch engine configuration has a 41.6% better (lower) average fetch power than a 2-way 164 KB instruction-cache-only fetch engine. The 279 KB trace cache fetch engine also exhibits better fetch energy, energy-delay, and energy-delay-squared results. For fetch energy, the 4-way set associative trace cache configuration uses 43.6% less energy than the 2-way 164 KB instruction-cache-only fetch engine. Similarly, energy-delay and energy-delay-squared for the 279 KB trace cache are also consistently lower than the 164 KB instruction-cache-only fetch engine. The energy-delay is 40.6% lower than the 164 KB instruction-cache-only configuration and the energy-delay-squared is 40.4% lower.

5.1.3 Discussion

Our results show that a larger fetch engine containing a trace cache can be more energy-efficient than a smaller instruction-cache-only fetch engine. In particular, the trace cache configurations exhibit lower average fetch power than the instruction-cache-only fetch engines due to conditional clock gating. This differs from the findings of Parikh *et al.* [17] for branch predictors. They found that power is directly proportional to branch predictor area. In contrast, the trace cache structures – although larger – are clock gated more often, yielding significant benefits in terms of energy savings compared to instruction-cache-only fetch engines.

Initially, it might seem that fetch engines which have trace caches are not very appealing when comparing to a realistic instruction-cache-only fetch engine like the Alpha 21364's 64 KB instruction cache plus 100 KB branch predictor, which corresponds to the 164 KB fetch engine area. For instance, the gain in performance (6.4% integer), when comparing the best trace cache data point to the 164 KB instruction-cache-only fetch engine area might not be seem worth the extra area. In the case of the floating point benchmarks the performance gain is negligible. Even when considering average fetch power, the best trace cache configurations exhibit very similar results as the 164 KB instruction-cache-only configuration. When examining fetch energy, a similar result is evident.

However, examining the energy-delay and energy-delay-squared results shows the energy-efficiency advantage of the trace cache. Even the smallest trace cache area, which is larger than the 164 KB fetch engine area, shows an improvement in energy-delay of 44.7% for integer and 36.4% for floating point and an improvement in energy-delay-squared of 47.8% for integer and 36.6% for floating point benchmarks. The improvements grow larger as the fetch engine area is increased.

This suggests that a larger trace cache-based fetch engine can be a better energy-efficient option if the chip area is not of utmost concern,even if a smaller instruction cache is lower-power.

5.2 Next Trace Predictor Area Sensitivity Exploration

Experiments to evaluate the sensitivity of the next trace predictor to changes in area were performed. Since the next trace predictor is composed of three main components (the correlating table, the secondary table, and the return history stack), three sets of experiments were performed in order to isolate the area sensitivity of each component. In each set of experiments, the trace cache was 128 KB, 4-way set associative with 16 instructions per line and a maximum of 6 branches per line. The outstanding trace buffer contained 128 entries. The results of each set of experiments is discussed below. Our results indicate that a hybrid next trace predictor with a 1 KB correlating table, 8 KB secondary table and 4-entry return history stack would likely yield the best power-performance-energy balance for our benchmark set.

5.2.1 Correlating Table Area Sensitivity

The first set of experiments varied the correlating table area while holding the area of the secondary table and the number of return history stack entries constant. A 32 KB secondary table and 128-entry return history stack entries were used, while the correlating table area was varied from 1 KB to 64 KB.

The performance results in Figure 7 for the correlating or primary table of the next trace predictor show that increasing the area of the primary table does not significantly improve the performance for the benchmarks we used in our experiments. Therefore, to get the best power-energy-performance, a correlating table size should be chosen which exhibits the best power and energy. Our results in Figures 8–11 indicate that a correlating table as small as 1 KB would be sufficient.

5.2.2 Secondary Table Area Sensitivity

The second set of experiments varied the secondary table area while holding the area of the correlating table and the number of return history stack entries constant. A 64 KB correlating table and a 128-entry return history stack were used, while the secondary table area was varied from 1 KB to 32 KB.

From Figure 7, we can see that the 8 KB secondary table area shows the most improvement in performance. So, we use this as a starting point to evaluate the best area when considering power and energy. The improvement in IPC between the 8 KB and 16 KB secondary table area is only 0.3% for the integer benchmarks and 0.03% for floating point benchmarks. Since this improvement is insignificant, the better secondary table area is the one with the better energy efficiency. Figures 8–11 show that the 8 KB secondary table has the best energy efficiency.

5.2.3 Return History Stack Sensitivity

The last set of experiments varied the number of return history stack entries while holding the areas of the correlating table and secondary table constant. A 64 KB correlating table and a 32 KB secondary table were used, while the number of return history stack entries was varied from 4 to 128 entries.

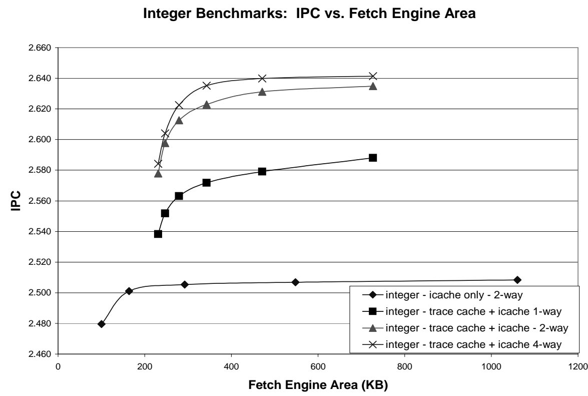
The results of these experiments showed that changing the number of return history stack entries did not affect performance significantly (See Figure 7). So, the best return history stack size depends on the power and energy results. Our results show that changing the number of return history stack entries does not significantly change the average fetch power, energy, energy-delay or energy-delay-squared. Therefore, the smallest possible return history stack of 4 entries would be the best choice, even though a return history stack of anywhere from 4 to 128 entries is reasonable.

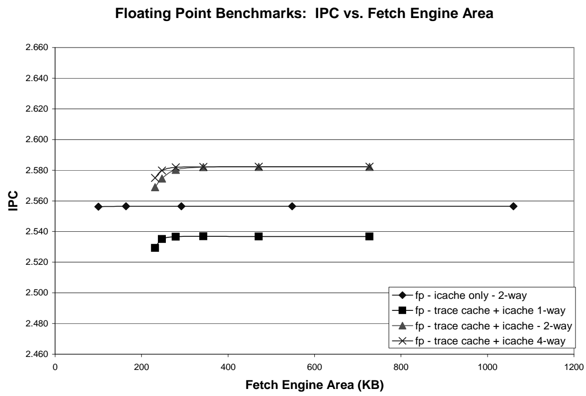
6 Conclusions

Our preliminary experiments show that sequential trace caches are highly energy and power-efficient while providing a significant performance improvement over fetch engines which consist of an instruction cache only. Although, at first glance, when examining performance and average fetch power, the trace cache does not appear to offer a significant benefit, when examining energy-delay and energy-delay-squared, it becomes evident that although a trace cache configuration may take up more chip area it yields better energy-efficiency overall.

Experimental results show that a 343 KB, 4-way set associative trace cache fetch engine outperforms a 292 KB instruction-cache-only fetch engine by 5% for integer benchmarks and 1% for floating point benchmarks. In addition, it does so using 68.3% less average fetch power, 70.3% less energy, 67.7% less energy-delay, and 69.6% less energy-delay-squared than a 292 KB, 2-way set associative instruction-cache-only fetch engine.

In addition, we find that for the Jacobson *et al.* hybrid path-based next trace predictor [13], a 1 KB correlating table, 8 KB secondary table, and 4-entry return history stack will likely provide the best power-performance-energy solution for our benchmarks.

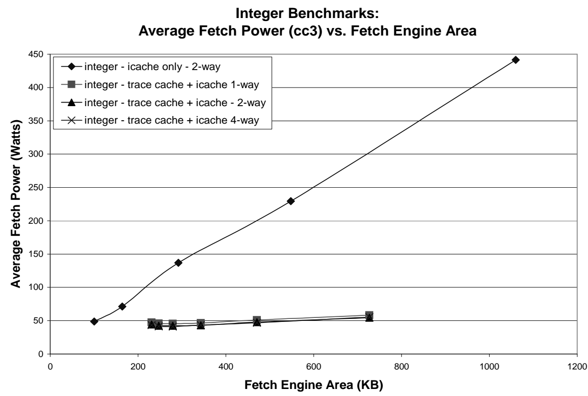


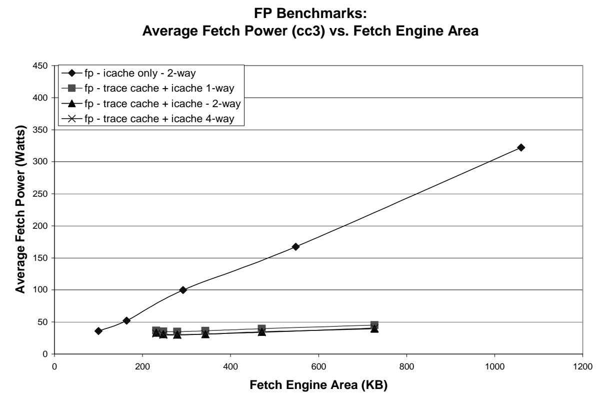


IPC: Integer Benchmarks				
Fetch Area	i$ only - 2-way	trace $ + i$ 1-way	trace $ + i$ - 2-way	trace $ + i$ 4-way
100 KB	2.47951			
164 KB	2.50103			
292 KB	2.50537			
548 KB	2.50690			
1060 KB	2.50833			
231 KB		2.53831	2.57786	2.58423
247 KB		2.55186	2.59764	2.60397
279 KB		2.56311	2.61253	2.62257
343 KB		2.57174	2.62293	2.63524
471 KB		2.57914	2.63113	2.63984
727 KB		2.58806	2.63493	2.64136

IPC: Floating Point Benchmarks				
Fetch Area	i$ only - 2-way	trace $ + i$ 1-way	trace $ + i$ - 2-way	trace $ + i$ 4-way
100 KB	2.55620			
164 KB	2.55644			
292 KB	2.55646			
548 KB	2.55646			
1060 KB	2.55646			
231 KB		2.52932	2.56886	2.57492
247 KB		2.53514	2.57462	2.57974
279 KB		2.53662	2.58020	2.58192
343 KB		2.53684	2.58200	2.58214
471 KB		2.53674	2.58214	2.58224
727 KB		2.53680	2.58216	2.58224

Figure 2: IPC vs. Fetch Engine Area

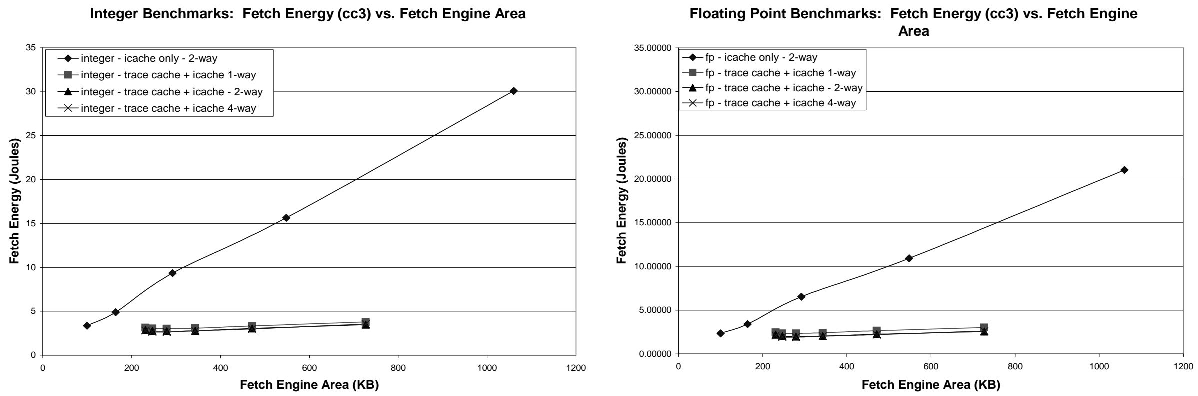




Average Fetch Power (W): Integer Benchmarks				
Fetch Area	i$ only - 2-way	trace $ + i$ 1-way	trace $ + i$ - 2-way	trace $ + i$ 4-way
100 KB	48.75053			
164 KB	71.38706			
292 KB	136.99947			
548 KB	229.41196			
1060 KB	441.30157			
231 KB		47.44414	44.35974	43.38843
247 KB		46.04020	42.54491	41.38283
279 KB		45.66034	42.31140	41.17559
343 KB		46.62791	43.22153	43.41110
471 KB		51.04313	47.72851	46.75357
727 KB		58.16180	54.40114	55.44836

Average Fetch Power (W): Floating Point Benchmarks				
Fetch Area	i$ only - 2-way	trace $ + i$ 1-way	trace $ + i$ - 2-way	trace $ + i$ 4-way
100 KB	35.92680			
164 KB	52.06942			
292 KB	99.93352			
548 KB	167.43488			
1060 KB	322.14374			
231 KB		36.64564	33.32412	32.02948
247 KB		35.11346	31.12586	29.87812
279 KB		34.89222	30.39364	29.59170
343 KB		36.23738	31.10198	31.41960
471 KB		39.69368	34.58582	33.95828
727 KB		45.08912	39.61218	40.37610

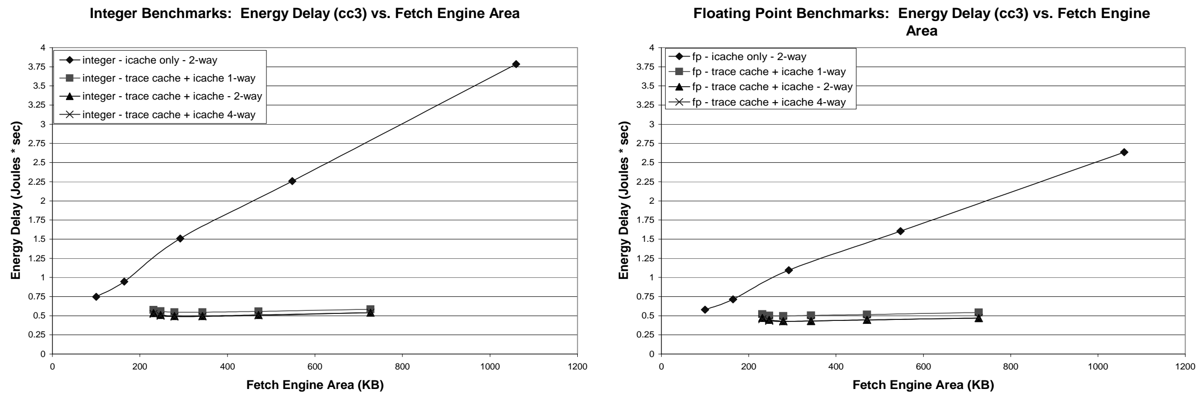
Figure 3: Average Fetch Power vs. Fetch Engine Area



Fetch Energy (J): Integer Benchmarks				
Fetch Area	i$ only - 2-way	trace $ + i$ 1-way	trace $ + i$ - 2-way	trace $ + i$ 4-way
100 KB	3.35600			
164 KB	4.87636			
292 KB	9.34650			
548 KB	15.64443			
1060 KB	30.07756			
231 KB		3.14151	2.88499	2.81256
247 KB		3.04050	2.75057	2.66953
279 KB		3.00773	2.72576	2.64220
343 KB		3.05773	2.77594	2.77437
471 KB		3.34070	3.05439	2.98303
727 KB		3.78956	3.47677	3.53481

Fetch Energy (J): Floating Point Benchmarks				
Fetch Area	i$ only - 2-way	trace $ + i$ 1-way	trace $ + i$ - 2-way	trace $ + i$ 4-way
100 KB	2.34792			
164 KB	3.40210			
292 KB	6.52848			
548 KB	10.93750			
1060 KB	21.04272			
231 KB		2.45494	2.19226	2.09232
247 KB		2.34740	2.03644	1.94092
279 KB		2.32938	1.97516	1.91740
343 KB		2.41962	2.01500	2.03488
471 KB		2.64904	2.23984	2.19880
727 KB		3.00462	2.56462	2.61336

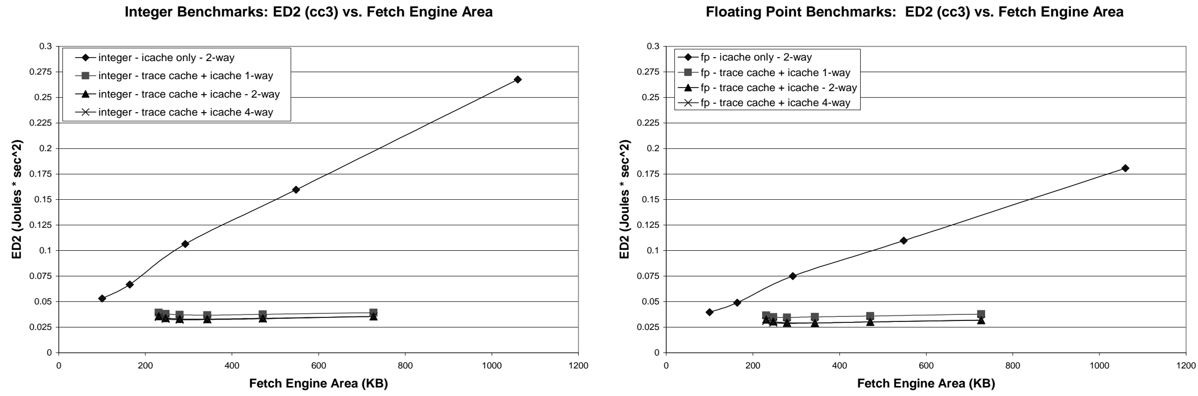
Figure 4: Fetch Energy vs. Fetch Engine Area



Energy Delay (J * sec): Integer Benchmarks				
Fetch Area	i$ only - 2-way	trace $ + i$ 1-way	trace $ + i$ - 2-way	trace $ + i$ 4-way
100 KB	0.7484			
164 KB	0.9467			
292 KB	1.5085			
548 KB	2.2590			
1060 KB	3.7858			
231 KB		0.5774	0.5341	0.5234
247 KB		0.5589	0.5091	0.4994
279 KB		0.5471	0.4974	0.4878
343 KB		0.5453	0.4968	0.4915
471 KB		0.5567	0.5112	0.5057
727 KB		0.5845	0.5395	0.5409

Energy Delay (J * sec): Floating Point Benchmarks				
Fetch Area	i$ only - 2-way	trace $ + i$1-way	trace $ + i$- 2-way	trace $ + i$ 4-way
100 KB	0.5781			
164 KB	0.7150			
292 KB	1.0951			
548 KB	1.6032			
1060 KB	2.6364			
231 KB		0.5211	0.4719	0.4550
247 KB		0.5023	0.4467	0.4321
279 KB		0.4954	0.4308	0.4249
343 KB		0.5035	0.4312	0.4318
471 KB		0.5154	0.4458	0.4448
727 KB		0.5422	0.4693	0.4721

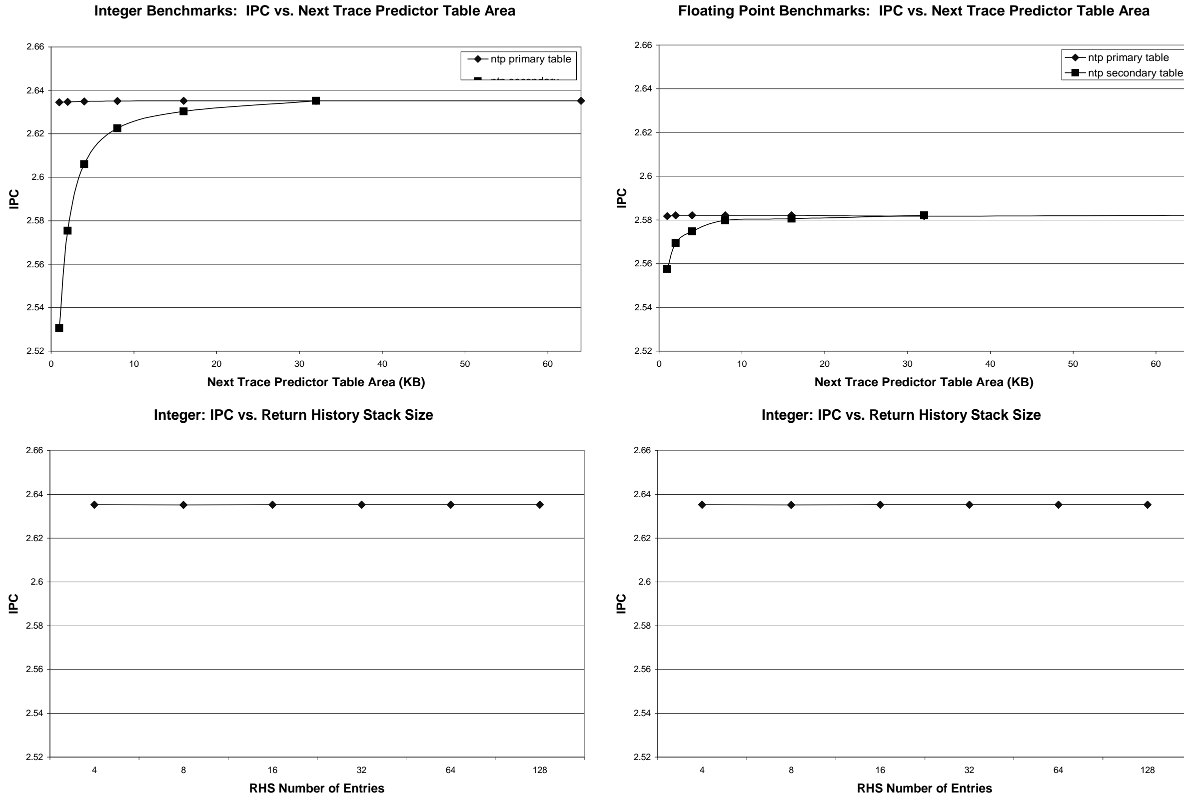
Figure 5: Energy Delay vs. Fetch Engine Area



ED2 (J * sec^2): Integer Benchmarks				
Fetch Area	i$ only - 2-way	trace $ + i$ 1-way	trace $ + i$ - 2-way	trace $ + i$ 4-way
100 KB	0.05307			
164 KB	0.0668			
292 KB	0.1065			
548 KB	0.1596			
1060 KB	0.2676			
231 KB		0.03934	0.03573	0.03487
247 KB		0.03798	0.03389	0.03314
279 KB		0.03711	0.03301	0.03222
343 KB		0.03686	0.03289	0.03240
471 KB		0.03759	0.03374	0.03331
727 KB		0.03927	0.03559	0.03561

ED2 (J * sec^2): Floating Point Benchmarks				
Fetch Area	i$ only - 2-way	trace $ + i$ 1-way	trace $ + i$ - 2-way	trace $ + i$ 4-way
100 KB	0.03962			
164 KB	0.04902			
292 KB	0.07504			
548 KB	0.10986			
1060 KB	0.18062			
231 KB		0.03654	0.03252	0.03108
247 KB		0.03502	0.0306	0.02932
279 KB		0.03448	0.02928	0.02878
343 KB		0.03508	0.02918	0.02924
471 KB		0.03592	0.03018	0.03010
727 KB		0.03778	0.03180	0.03196

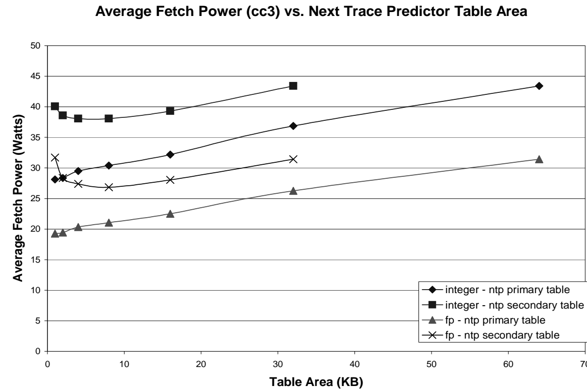
Figure 6: Energy Delay Squared vs. Fetch Engine Area



Integer: IPC							
Table Type	1 KB	2 KB	4 KB	8 KB	16 KB	32 KB	64 KB
NTP primary table	2.6345	2.6347	2.6349	2.6352	2.6352	2.6352	2.6352
NTP secondary table	2.5306	2.5754	2.6061	2.6225	2.6304	2.6352	
Table Type	4 entry	8 entry	16 entry	32 entry	64 entry	128 entry	
NTP RHS	2.6353	2.6352	2.6352	2.6352	2.6352	2.6352	

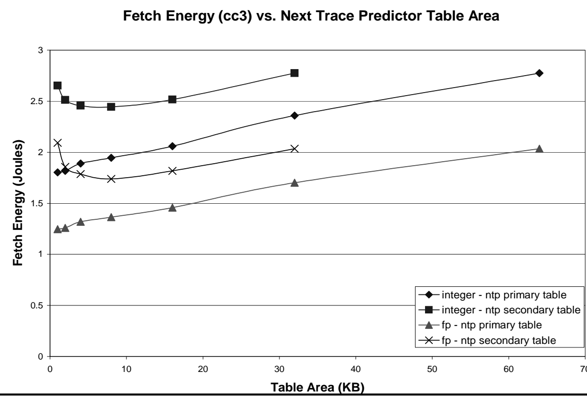
FP: IPC							
Table Type	1 KB	2 KB	4 KB	8 KB	16 KB	32 KB	64 KB
NTP primary table	2.5817	2.5821	2.5821	2.5821	2.5821	2.5817	2.5821
NTP secondary table	2.5576	2.5695	2.5748	2.5798	2.5806	2.5821	
Table Type	4 entry	8 entry	16 entry	32 entry	64 entry	128 entry	
NTP RHS	2.5821	2.5821	2.5821	2.5821	2.5821	2.5821	

Figure 7: Next Trace Predictor Sensitivity Study – IPC vs. NTP Table Size and RHS Size



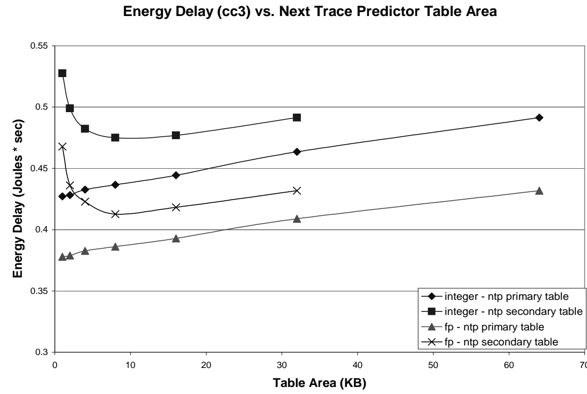
Average Fetch Power (W): Next Trace Predictor Tables							
	1 KB	2 KB	4 KB	8 KB	16 KB	32 KB	64 KB
Integer - NTP primary table	28.1214	28.3489	29.4961	30.3892	32.1699	36.8809	43.4111
FP - NTP primary table	19.2480	19.4330	20.3417	21.06590	22.4921	26.2659	31.4196
Integer - NTP secondary table	40.0667	38.5924	38.0873	38.0887	39.3145	43.4111	
FP - NTP secondary table	31.7099	28.3564	27.3863	26.8370	28.0530	31.4196	

Figure 8: Average Fetch Power vs. Next Trace Predictor Table Area



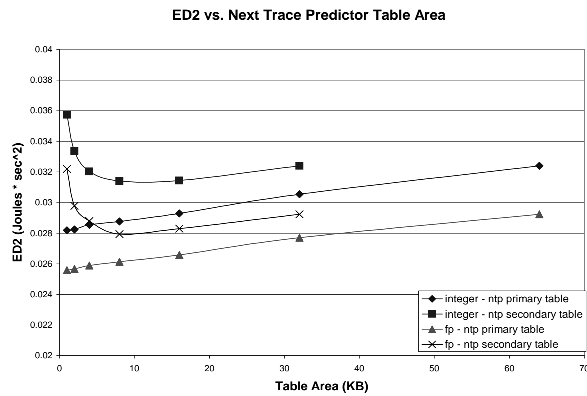
Fetch Energy: Next Trace Predictor Tables							
	1 KB	2 KB	4 KB	8 KB	16 KB	32 KB	64 KB
Integer - NTP primary table	1.80217	1.81641	1.88917	1.94573	2.05893	2.35866	2.77377
FP - NTP primary table	1.2474	1.25942	1.31824	1.36504	1.45732	1.70142	2.03488
Integer - NTP secondary table	2.65194	2.51265	2.45670	2.44437	2.51634	2.77437	
FP - NTP secondary table	2.09232	1.85644	1.78642	1.73794	1.81666	2.03488	

Figure 9: Fetch Energy vs. Next Trace Predictor Table Area



	1 KB	2 KB	4 KB	8 KB	16 KB	32 KB	64 KB
Integer - NTP primary table	0.4272	0.4281	0.4327	0.4366	0.4444	0.4634	0.4915
FP - NTP primary table	0.3780	0.3789	0.3827	0.3861	0.3928	0.4089	0.4318
Integer - NTP secondary table	0.5276	0.4989	0.4823	0.4751	0.4770	0.4915	
FP - NTP secondary table	0.4678	0.4360	0.4227	0.4126	0.4182	0.4318	

Figure 10: Energy Delay vs. Next Trace Predictor Table Area



	1 KB	2 KB	4 KB	8 KB	16 KB	32 KB	64 KB
Integer - NTP primary table	0.02819	0.02824	0.02854	0.02877	0.02929	0.03054	0.03240
FP - NTP primary table	0.02558	0.02566	0.02588	0.02614	0.02658	0.0277	0.02924
Integer - NTP secondary table	0.03574	0.03335	0.03203	0.03141	0.03144	0.03240	
FP - NTP secondary table	0.03220	0.02978	0.02880	0.02794	0.0283	0.02924	

Figure 11: Energy Delay Squared vs. Next Trace Predictor Table Area

Future directions of this work include exploring the power, performance, and energy characteristics of other next trace prediction mechanisms. In addition, we will add data points for other instruction cache areas and associativities. We also plan to evaluate other trace cache implementations such as the block-based trace cache [3], dynamic prediction directed trace cache [9] and the filter trace cache [25].

Acknowledgements

This work is supported in part by the National Science Foundation under grant nos. CCR-0133634, CCR-0105626, and EIA-0224434, and a grant from Intel MRL. We would also like to thank Eric Rotenberg, Dee Weikle, and Jason Hiser for their helpful input.

References

[1] R. Iris Bahar, G. Albera, and S. Manne. Power and performance tradeoffs using various caching strategies. In *Proceedings of the 1998 International Symposium on Low Power Electronics and Design*, pages 64–69. ACM Press, 1998.

[2] P. Bannon. Personal communication with K. Skadron, Sep. 2002.

[3] B. Black, B. Rychlik, and J.P. Shen. The block-based trace cache. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 196–207. IEEE Computer Society Press, May 1999.

[4] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 83–94, June 2000.

[5] D. C. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. *Computer Architecture News*, 25(3):13–25, June 1997.

[6] T. Conte, K. Menezes, P. Mills, and B. Patel. Optimization of instruction fetch mechanisms for high issue rates. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 333–344, Oct 1996.

[7] S. Dutta and M. Franklin. Control flow prediction with tree-like subgraphs for superscalar processors. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 258–263, Nov 1995.

[8] D.H. Friendly, S.J. Patel, and Y.N. Patt. Alternative fetch and issue policies for the trace cache fetch mechanism. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 24–33, 1997.

[9] J.S. Hu, N. Vijaykrishnan, M. J. Irwin, and M. Kandemir. Using dynamic branch behavior for power-efficient instruction fetch. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI 2003)*, 2003.

[10] J.S. Hu, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. Power-efficient trace caches. In *Proceedings of the 2002 Design, Automation and Test in Europe Conference and Exhibition (DATE '02)*, 2002.

[11] J. W. Haskins, Jr. and K. Skadron. Memory reference reuse latency: Accelerated sampled microarchit ecture simulation. In *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Syste ms and Software*, pages 195–203, Mar. 2003.

[12] Q. Jacobson, S. Bennett, N. Sharma, and J.E. Smith. Control flow speculation in multiscalar processors. In *Proceedings of the Third International Symposium on High-Performance Computer Architecture*, pages 218–229, Feb. 1997.

[13] Q. Jacobson, E. Rotenberg, and J. E. Smith. Path-based next trace prediction. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 14–23, 1997.

[14] J. Johnson. Expansion caches for superscalar processors. Technical Report CSL-TR-94-630, Computer Science Laboratory, Stanford University, June 1994.

[15] N.S. Kim, K. Flautner, D. Blaauw, and T. Mudge. Drowsy instruction caches: leakage power reduction using dynamic voltage scaling and cache sub-bank prediction. In *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 219–230. IEEE Computer Society Press, 2002.

[16] J. Montanaro and et al. A 160-mhz 32-b 0.5-w cmos risc processor. Technical Report Vol. 9, Digital Technical Journal, Digital Equipment Corporation, 1997.

[17] D. Parikh, K. Skadron, Y. Zhang, M. Barcella, and M. St an. Power issues related to branch prediction. In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture*, pages 233–44, Feb. 2002.

[18] A. Peleg and U. Weiser. Dynamic flow instruction cache memory organized around trace segments independent of virtual address line. In *U.S. Patent Number 5,381,533*, Jan. 1995.

[19] E. Rotenberg, S. Bennett, and J. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. Technical Report 1310, Cs Dept. University of Wisconsin, Madison, April 1996.

[20] E. Rotenberg, S. Bennett, and J. E. Smith. A trace cache microarchitecture and evaluation. *IEEE Transactions on Computers*, 48(2):111–120, 1999.

[21] E. Rotenberg, S. Bennett, and J.E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 24–35, 1996.

[22] A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud. Multiple block ahead branch predictors. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct 1996.

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

[24] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware microarchitecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 2–13, June 2003.

[25] W. Tang, R. Gupta, and A. Nicolau. Design of a predictive filter cache for energy savings in high performance processor architectures. In *Proceedings of the 2001 International Conference on Computer Design*, pages 68–73, 2001.

[26] T.-Y. Yeh, D.T. Marr, and Y.N. Patt. Increasing the instruction fetch rate via multiple branch prediction and a branch address cache. In *Proceedings of the 7th International Conference on Supercomputing*, pages 67–76, July 1993.

[27] Y. Zhang and J. Yang. Low cost instruction cache designs for tag comparison elimination. In *Proceedings of the 2003 International Symposium on Low Power Electronics and Design*, pages 266–269. ACM Press, 2003.