Evaluating Trace Cache Energy Efficiency

MICHELE CO, DEE A. B. WEIKLE, and KEVIN SKADRON
University of Virginia

Future fetch engines need to be energy efficient. Much research has focused on improving fetch bandwidth. In particular, previous research shows that storing concatenated basic blocks to form instruction traces can significantly improve fetch performance. This work evaluates whether this concatenating of basic blocks translates to significant *energy-efficiency* gains. We compare processor performance and energy efficiency in trace caches compared to instruction caches. We find that, although trace caches modestly outperform instruction cache only alternatives, it is branch-prediction accuracy that really determines performance and energy efficiency. When access delay and area restrictions are considered, our results show that sequential trace caches achieve very similar performance and energy efficiency results compared to instruction cache-based fetch engines and show that the trace cache's failure to significantly outperform the instruction cache-based fetch organizations stems from the poorer implicit branch prediction from the next-trace predictor at smaller areas. Because access delay limits the theoretical performance of the evaluated fetch engines, we also propose a novel ahead-pipelined next-trace predictor. Our results show that an STC fetch organization with a three-stage, ahead-pipelined next-trace predictor can achieve 5–17% IPC and 29% ED^2 improvements over conventional, unpipelined organizations.

Categories and Subject Descriptors: C.4 [**Performance of Systems**]: Design Studies

General Terms: Measurement, Performance

Additional Key Words and Phrases: Trace cache, fetch engine energy efficiency

1. INTRODUCTION

Energy efficiency has become important for almost all new chip designs. The fetch unit contributes a large portion of total power consumption in a microprocessor. For example, Bose et al. [2002] measure the POWER4's front-end instruction delivery path to consume 20% of net chip system power and about 35% of the processor core. Research trends also point toward aggressive fetch engines for maximum performance. Understanding how fetch organization affects processor energy efficiency is important to processor design.

The fetch unit's role is to feed the dynamic instruction stream to the execution unit. Instruction caches store instructions in static program order.

Authors' address: M. Co, D. A. B. Weikle, and K. Skadron, Department of Computer Science, School of Engineering, University of Virginia 151 Engineer's Way, P.O. Box 400740 Charlottesville, Virginia 22904-4740; email: {micheleco,dweikle,skadron}@cs.virginia.edu.

Because of the presence of taken control-flow instructions, some of the instructions fetched from the instruction cache are unused. Trace caches are one type of high-fetch bandwidth mechanism, which instead stores instructions in dynamic program order, stitching together several nonsequential basic blocks and increasing effective fetch bandwidth. The high-performance benefits of many high-fetch bandwidth mechanisms have been evaluated in the research community.

However, many 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 and many of these mechanisms have not been evaluated in terms of energy efficiency.

Most trace-cache implementations [Johnson 1994; Peleg and Weiser 1995; Rotenberg et al. 1996a, 1996b] do not suffer from these complexity drawbacks and thus have advantages in fetch engine design. Trace-cache implementations have been evaluated for their performance benefits over instruction cache-only fetch designs. Trace-caches have also been evaluated for energy efficiency within the trace cache design space, but we are not aware of any work analyzing the relative energy efficiency of trace caches compared to instruction cache-only fetch organizations.

Our work models several types of trace caches: the conventional or concurrent trace cache (CTC), in which trace cache and instruction cache are probed in parallel, the sequential trace cache (STC), described by Rotenberg et al. [1996a, 1996b], which accesses the trace cache and instruction cache sequentially, and the block-based trace cache (BBTC), proposed by Black et al. [1999], which stores basic blocks for assembly into traces. We present four sets of experimental results.

First, we compare fetch-engine organizations without area budget restrictions across the parameters of associativity, area, and trace length. We evaluate the effect of these parameters on overall performance and energy efficiency. Then, to explore intrinsic branch-prediction capabilities, we evaluate all fetch organizations with a similar predictor based on the *next-trace predictor* (NTP) [Jacobson et al. 1997]. To account for the trend of decreasing access times resulting from high clock speeds, we reevaluate the fetch engines where the area of each component in the fetch engine organization is restricted. Finally, we introduce and evaluate an ahead-pipelined NTP to address decreasing cycle times. Each comparison is made with respect to two parameters: performance (IPC) and energy-delay squared (ED^2) [Zyuban and Strenski 2002].

This work shows that branch prediction is much more important than the storage order of instructions, either in a trace cache or in an instruction cache. Providing fewer misspeculated instructions to the execution engine improves overall processor energy efficiency, but newer, improved branch predictor designs make the tradeoff between trace caches and instruction caches less clear.

The rest of the paper is organized as follows: Section 2 presents related work, Section 3 presents experimental methodology, Sections 4 and 5 present

experimental results, and Section 6 presents conclusions and directions for future work.

2. RELATED WORK

2.1 Fetch Engine Design

Friendly, Patel et al. [1997] and Rotenberg et al. [1996a, 1996b, 1999] performed comprehensive studies of the trace cache design space with respect to performance. We perform a similar design-space study to evaluate power, energy, and performance tradeoffs on a more current processor pipeline.

Research has explored ways to reduce the power dissipation of trace caches. Hu et al. [2002] showed that sequentially accessing the trace cache and the level 1 (L1) instruction cache has significant power savings over accessing the two structures simultaneously. In subsequent work, Hu et al. [2003] also compared the concurrent access trace cache (CTC), sequential trace cache (STC), and a new design, the dynamic direction prediction-based trace cache (DPTC) for power efficiency and performance. They found that the DPTC exhibits less performance loss than the STC, but with similar power consumption. Our work expands this previous work by providing a comparison of fetch units containing either CTCs, STCs, or BBTCs to fetch units containing an instruction cache-only. We also evaluate the relative contribution of trace length and branch prediction in overall processor energy efficiency and determine the effect of additional parameters, such as leakage and delay.

Bahar et al. [1998], Kim et al. [2002], and Zhang and Yang [2003] have performed work to improve traditional instruction cache energy consumption without adversely affecting processor performance or on-chip energy consumption. Bahar suggested the use of buffers between the L1 and L2 caches to improve cache energy efficiency. Kim introduced the drowsy-instruction cache, which selectively powered off segments of the instruction cache to improve power consumption. Zhang proposed tag comparison elimination for reducing cache energy consumption. Our work focuses on evaluating the relative energy efficiency of high-fetch bandwidth fetch organizations compared to those with instruction caches only as opposed to focusing on techniques to improve traditional instruction cache energy efficiency.

Solomon et al. [2001] introduced the microoperation cache (μC) as an alternative front end for the Intel P6 processor family. The μC stores basic blocks in decoded μop form and provides similar fetch bandwidth at lower power consumption. The focus of their work was not to increase fetch bandwidth, but rather to find a more energy efficient fetch engine design with comparable performance for the Intel IA-32 architecture. The goal of our work is to understand the characteristics that affect the energy efficiency of trace caches in a relatively architecturally independent way. The IA-32 architecture's front-end characteristics are unique and insights drawn from evaluating the μC are not easily extended to architectures, which support other ISAs. An evaluation of the μC, while interesting, is not included because this trace cache design may be an artifact of the high cost of decoding x86 instructions. Instead, we chose

to provide more general insights with respect to the relative energy efficiency of trace cache-based fetch organizations versus instruction cache-only fetch organizations.

Parikh et al. [2002] explored the role of branch predictor organization on power, energy, and performance tradeoffs for fetch engine design. They found that although extra power might need to be expended to improve branch prediction accuracy, overall processor power and energy dissipation can be reduced. Our work focuses more broadly on both the cache and prediction mechanisms in various fetch engine organizations and on their ultimate impact on overall processor energy efficiency.

Fahs et al. [2001] proposed the rePLay microarchitecture, which includes hardware support for dynamic optimization. The rePLay framework consists of a frame constructor, programmable optimization engine, frame cache, sequencing component, and misspeculation recovery mechanism. The frame cache is similar to trace caches in that it stores variable-length sequences of instructions that may span multiple traditional cache lines. Frames differ from traces in that frames are atomic regions of instructions, which contain no internal control dependencies, and in that there is no notion of partial commit of a frame. All instructions within a frame must execute successfully to commit or a full rollback must occur. Hardware support for dynamic optimization is a field that requires the processor architecture to support hardware/software communication. Exploring the energy efficiency of architectures, which support software-controlled hardware-performed dynamic optimizations, is a very interesting field in its own right. However, including an evaluation of this class of architectures adds many more evaluation parameters, which make discerning valuable insights about the relative energy efficiency of purely hardware-based fetch organizations more difficult, so is not included.

Rosner et al. [2004] present the PARROT microarchitecture, which utilizes trace caching, dynamic optimizations, and pipeline decoupling for improving performance with reduced energy consumption. The PARROT microarchitecture is designed to take advantage of the paradigm that 90% of execution time is spent in 10% of the static code. This microarchitecture has the ability to identify hot (frequently executed) and cold (infrequently executed) traces and send them to devoted hot and cold execution pipelines, respectively. Traces designated to the hot pipeline are aggressively optimized. Their results show significant performance gain and energy savings. The focus of this paper is to compare trace cache-based fetch organizations to conventional instruction cache fetch organizations. A fair evaluation of the class of aggressive architectures, which include dynamic optimization hardware such as the PARROT, rePLay, and many other architectures, involves many additional parameters and is not included here.

2.2 Fetch Bundles [Stream Fetching/Implicit Multiple-Branch Prediction]

Yeh and Patt [1992] introduced the basic block target buffer to reduce the delay in predicting branch target addresses. They reduced the delay by storing both the target and fall-through addresses of each basic block predicted.

Reinman et al. [1999] expanded on Yeh and Patt's basic block target buffer work [Yeh and Patt 1992] and introduced the fetch target buffer (FTB) to decouple fetching from fetch predictions. They introduced the idea of terminating fetch bundles with strongly biased taken branches instead of terminating a fetch bundle on the first encountered branch. In other words, one fetch bundle represents a sequence of instructions whose internal branches are implicitly predicted as not taken. The final branch in the block is still predicted explicitly by a branch predictor to determine the start address of the next fetch bundle. Ramirez et al. [2002] introduce a fetch model, which extends Reinman's work by using streams, a stricter definition of Reinman's fetch bundle. A stream is an arbitrary length sequence of instructions (including not taken branches) terminating with a taken branch. Therefore, branch direction predictions are implicit within a stream. This stream definition means that the stream predictor produces an explicit address prediction of the last branch in the previous stream.

The instruction cache *STREAM* model described in Section 3.1 uses some of the ideas of Reinman's and Ramirez' work. We use Ramirez' stream definition to allow fetching past not taken branches. To make more straightforward comparisons across fetch engine designs and to focus on the benefits of instruction cache versus trace cache, we do not decouple fetch predictions from instruction fetching as both Reinman and Ramirez do.

Oberoi and Sohi [2003] proposed *parallelism in the front end*, in which several instruction sequence fragments are fetched and renamed in parallel from a banked instruction cache. Our experiment to isolate the effects of branch prediction on fetch engine designs evaluates a fetch organization with an instruction cache and a next trace predictor, which is a simplified, sequential version of the Oberoi work. A full evaluation of parallelized trace construction fetch organizations is beyond the scope of this work. In this work, we instead focus on understanding the energy-efficiency implications of sequential fetch organizations.

Several high-fetch bandwidth mechanisms, such as branch address cache [Yeh et al. 1993], subgraph predictor [Dutta and Franklin 1995], collapsing buffer [Conte et al. 1996], multiple-block ahead predictor [Seznec et al. 1996], block-based trace cache [Black et al. 1999], and trace cache [Johnson 1994; Peleg and Weiser 1995; Rotenberg et al. 1996a, 1996b] have been proposed. Many of these mechanisms have drawbacks in terms of complexity and power. Therefore, for this work, we only consider the trace cache described by Rotenberg et al. [1996b] and the block-based trace cache described by Black et al. [1999].

There is separate previous work evaluating the relative performance of trace caches compared to instruction caches, evaluating hardware optimizations for improving instruction cache energy efficiency, and evaluating the relative energy efficiency of particular trace cache designs. We are not aware of any further research examining 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.

Table I. Simulated Processor Microarchitecture

Processor Core	
Active list	128 entries
Physical registers	80
LSQ	128 entries
Issue width	16 instructions per cycle
Functional units	16 IntALU, 4 IntMult/Div, 8 FPALU, 4 FPMult/Div, 2 memory 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 2-way interleaved both 2-cycle latency
L2	Unified, 4 MB, 8-way LRU, 128B blocks, 12-cycle latency, writeback
Memory	225 cycles (75 ns)
TLB size	128-entry, fully assoc., 30-cycle miss penalty
Branch Predictor	
Branch predictor	Hybrid PAg/GAg with GAg chooser
BTB	2 K-entry, 2-way
RAS	32-entry

3. EXPERIMENTAL METHODOLOGY

All experiments in this work use the SimpleScalar Toolset [Burger and Austin 1997] and a modified Wattch [Brooks et al. 2000] infrastructure with a power model based on the Alpha 21364 [Skadron et al. 2003]. The base out-of-order simulator was extended to include concurrent trace cache (CTC), sequential trace cache (STC), block-based trace cache (BBTC), and path-based NTP models. The microarchitecture model is summarized in Table I. Cache and predictor acronyms are listed in Table II and evaluated fetch engines are listed in Tables III and IV.

To more closely study the efficiency of the fetch engines, we chose a highly parallelizing execution core. We altered the base microarchitecture to have 128 instruction fetch queue entries, 128 register rename entries, and 128 load/store queue entries. In addition, the base architecture was modified so that 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. Highly parallelizing execution cores have been used in prior trace cache studies [Rotenberg et al. 1996b, 1997, 1999; Black et al. 1999]. We, therefore, use a similar wide core here. Our results show that branch prediction accuracy has a strong effect on the performance of the wider cores.

Since current CPU designs are increasingly using conditional clocking techniques to reduce power consumption, the power and energy metrics were calculated using Wattch's conditional clocking method, which scales power linearly with port or unit usage [Brooks et al. 2000].

Table II. Cache and Predictor Acronyms

Acronym	Cache	Acronym	Predictor
IC	Instruction cache	HYB_BPRED	Hybrid branch predictor
STC	Sequential access trace cache	GPERC	Global perceptron branch predictor
CTC	Concurrent access trace cache	NTP	Path-based next-trace predictor
BBTC	Block-based trace cache	TT	Trace table

Table III. Summary of Initial Fetch Engines Evaluated

Original Fetch Engines Evaluated			
Fetch Engine Name	Cache	Predictor	Fetch Bundle
STC_NTP	STC w/backing IC	NTP w/backing HYB_BPRED	Trace
CTC_NTP	CTC w/backing IC	NTP w/backing HYB_BPRED	Trace
IC_CLASSIC_BPRED	IC	HYB_BPRED	Cache line
IC_CLASSIC_GPERC	IC	GPERC	Cache line
IC_STREAM_BPRED	IC	HYB_BPRED	Stream
IC_STREAM_GPERC	IC	GPERC	Stream
BBTC_TT	BBTC	TT	4-Block trace

Table IV. Summary of Fetch Engines Evaluated to Isolate Branch Prediction Effects

Additional Fetch Engines Evaluated			
Fetch Engine Name	Cache	Predictor	Fetch Bundle
IC_CLASSIC_NTP	IC	NTP	Cache line
IC_STREAM_NTP	IC	NTP	Stream
BBTC_NTP	BBTC	NTP	4-Block trace

To model leakage, when the port or unit is not in use, a fixed ratio of maximum power dissipation is charged: 10% in most experiments. We examined the effect of increasing leakage, but found that leakage differences are minimal between the various fetch engines when the area of each fetch component is restricted. (See Section 4.3.1 for further discussion.)

We conducted several experiments to evaluate the performance and energy efficiency of some common fetch engine designs. Table II describes the acronyms used in this paper and Table III describes the initial set of fetch engines evaluated. We then performed a set of controlled experiments to determine if branch prediction accuracy plays a major role in the difference between the evaluated fetch engines. To isolate the effect of differing branch prediction accuracy, we evaluate each fetch organization when the best-performing branch predictor, the NTP, is used in the design. Table IV lists these additional fetch organizations. The effect of delay is considered in another set of experiments by restricting the area of each fetch component and a novel ahead-pipelined NTP, which improves NTP performance in the face of delay, is evaluated. Sections 3.1 and 3.2 discuss the details of the fetch organizations we evaluated.

3.1 Instruction Cache Model

For a more direct comparison to the trace cache fetch engines, the instruction caches modeled in our experiments have long cache lines (16 instructions wide) and are two-way interleaved.

We simulate two types of fetching behavior for the instruction cache modeled in our experiments: *CLASSIC* and *STREAM*.

3.1.1 *IC_CLASSIC.* The *CLASSIC* fetch behavior modeled represents the capability of current instruction cache fetch engines. The branch predictor is allowed to predict only the first branch encountered per fetch bundle and a maximum of 16 instructions may be fetched.

3.1.2 *IC_STREAM.* The *STREAM* model we use is a simple, but aggressive approximation of stream fetching. For consistency with the other fetch organizations evaluated, our model differs from previous work. We use a two-way interleaved instruction cache, which allows the fetch engine to fetch up to the first taken branch or a maximum of 16 instructions.

Our model does not use a basic block target buffer as in Yeh and Patt's work [1992]. Instead, a traditional branch target buffer and branch predictor is used to make branch target address predictions. Because the trace cache fetch engines we evaluated do not decouple fetch predictions from instruction fetching, we chose to evaluate our instruction cache fetch engines without decoupling. This differs from the decoupled fetch designs of Reinman et al. [1999] and Ramirez et al. [2002]. However, our *STREAM* model uses Ramirez' definition of stream (sequential list of instructions terminated by a taken branch).

Our IC_STREAM model fetches aggressively as a result of the two-way interleaved instruction cache. We also allow the branch predictor to predict multiple-branches within a single cycle. We believe that this model is useful for comparison because it represents the potential for an instruction cache-based fetch engine with idealized fetch bandwidth and with multiple-branch prediction capabilities. We chose this design to evaluate instruction cache fetch engines with more idealized (larger) fetch bundles.

3.2 Trace Cache Models

3.2.1 *Sequential and Concurrent Access Trace Caches.* The sequential trace cache (STC) modeled in the experiments is the model described by Rotenberg et al. [1999] shown in Figure 1. The STC consists of a path-based NTP [Jacobson et al. 1997], which predicts the next-trace to be fetched, outstanding trace buffers (OTB) to hold in-flight predicted traces, and the trace cache itself. Instructions are stored in undecoded form unlike the μop cache present in the Intel Pentium 4 architecture. Sequential access was modeled as described in the work of Hu et al. [2002]. The backing instruction cache is only probed on the next cycle after a trace cache miss. The STC's power was modeled as an array structure, similar to an instruction cache, with one read and one write port. The concurrent access trace cache (CTC) is the same fetch organization as the STC, except that the backing instruction cache and the trace cache

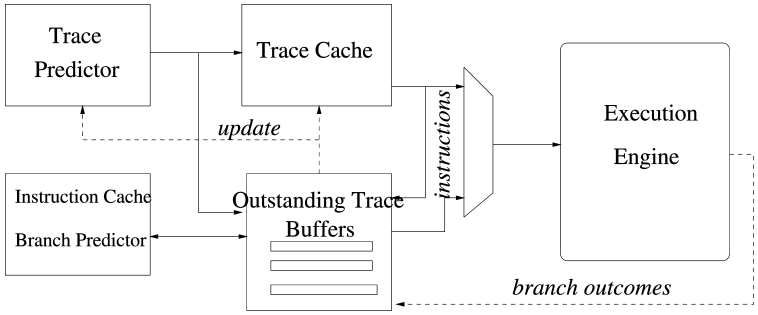


Fig. 1. STC (and CTC) model. (Patterned after figure in Rotenberg et al. [1999]).

are probed in parallel. The power model for the CTC is adjusted to reflect the parallel trace cache and instruction cache access.

Traces may be defined in many ways. Since we use Jacobson et al. hybrid NTP [1997], we use the definition of trace used in their work for all CTC and STC simulations. A trace has a maximum of 16 instructions and as many as seven branches (six internal branches, plus a possible seventh terminating branch). Indirect branches terminate a trace. The NTP uses path history information (recently committed traces) to make predictions much like a global history branch predictor (GAs). This information is combined with trace history to index a table that makes a prediction about the next-trace to be fetched. In our experiments, eight previous trace identifiers are hashed together to get indexes into the 64-K entry correlating table and into the 32-K entry secondary table. A selector mechanism chooses the prediction from the more accurate table.

To model the power of the hybrid NTP, the correlating table, secondary table, return history stack (RHS), and path history register are each modeled as array structures with one read and one write port.

The outstanding trace buffer (OTB) maintains information about in-flight traces. When an entire trace commits, the trace is written to the trace cache (if needed) and the OTB entry is reclaimed. OTB entries also maintain information needed to recover from mispredicted branches. The power for the OTB is modeled as an array structure with two read ports and one write port. One read port is shared by fetch and misprediction recovery mechanisms and one read port is devoted to the commit time mechanism. The single write port is shared between fetch and misprediction recovery mechanisms. The experiments in Section 4 use 128 OTB entries.

3.2.2 *Block-Based Trace Cache Model.* The block-based trace cache (BBTC) described by Black et al. [1999] (Figure 2) is another type of trace cache, which represents a trace as a series of pointers to basic blocks, which are stored in a block cache. The BBTC modeled in our experiments consists of a trace table, which makes next-trace predictions, a block cache, which stores basic blocks for trace construction, a rename table to maintain fetch address renaming, and a fill unit, which controls the update of the other three components. The trace table predicts a series of blocks to fetch using block identifier execution history and branch history bits. These predicted blocks are fetched

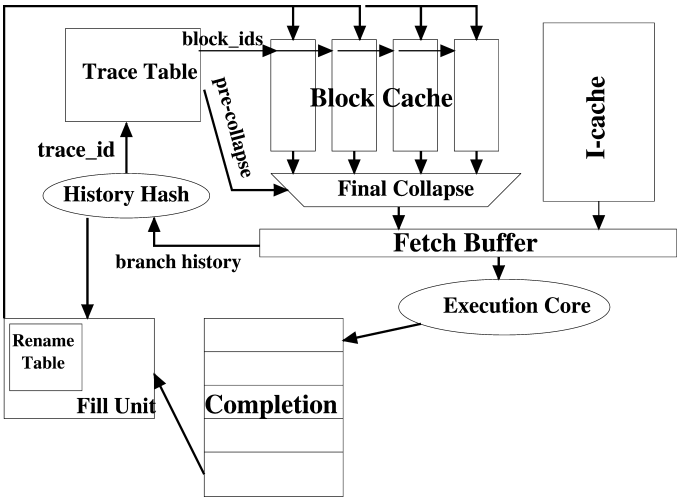


Fig. 2. BBTC model patterned after [Black et al. 1999].

from the block cache and assembled to construct a trace. Blocks are allocated to the block cache by the rename table, which maintains a mapping of fetch addresses to block identifiers. The fill unit controls the update of the trace table, block cache, and rename table.

The trace definition in Black et al. BBTC model is different than the STC and CTC model's trace definition. A BBTC trace is defined to be a series of blocks with each block being defined as a series of instructions terminated by a branch, or a fixed user-defined maximum number of sequential instructions. There are no other special trace termination conditions. To make the BBTC trace definition more comparable to the STC trace definition, we altered the BBTC trace definition to terminate traces on indirect branches, as in Rotenberg's work. Our experiments show that this modification in the trace definition improves the BBTC's trace predictions and consequently improves IPC. We chose to model a replication of four and a maximum basic block size of six instructions to match the published best-performing BBTC.

For power modeling, the BBTC components are each modeled as array structures with one read port and one write port.

3.3 Cache Parameters

The fetch engine experiments, which contain either CTC, STC, or BBTC, also include a noninterleaved instruction cache, which serves as backup in the case of a trace cache miss. The fetch engine components, which were held constant, are shown in Table V.

3.3.1 *Trace Cache Parameters.* Since the number of components in the CTC, STC, and BBTC designs differ from the number of components in IC designs, an equal-area comparison is difficult. Therefore, we examine the fetch engines over a range of different fetch engine areas. We first evaluate the fetch engines when the area of individual fetch engine components is unrestricted.

Table V. Parameters Held Constant for STC, CTC, and BBTC Experiments

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

Table VI. CTC/STC and IC Areas with Corresponding Fetch Engine Areas[a]

Fetch Engine Area (KB)	CTC/STC Area (KB)	Fetch Engine Area (KB)	IC Area (KB)
980	16	100	64
996	32	164	128
1028	64	292	256
1092	128	548	512
1220	256	1060	1024
1476	512		

[a]Used in experiments which use default NTP and OTB components. Cache area is included in the fetch engine area total.

This allows us to examine the theoretical potential of the various fetch engines. Then, to consider access time, we restrict the area of each individual component of each fetch engine to areas of 2 through 512 KB in successive simulations.

Associativities for the STC and CTC are varied in the experiments, but the replacement policy is fixed to LRU, and the line size is fixed to the length of one trace (16 instructions). Table VI shows the fetch engine areas used in the experiments of Section 4. These experiments use the ideal NTP and OTB parameters specified by Jacobson et al. [1997]. The area used for the CTC/STC alone is listed alongside the total fetch engine area. The remaining fetch engine area is calculated by totaling the area of the backing instruction cache, branch predictor (including BTB), OTB, and hybrid NTP.

In the first comparison, the area of the STC is varied while the areas of the other components are held constant (see Table V). In a second comparison, the areas of the STC, NTP, and OTB are limited to 2 through 512 KB. Similarly, the associativities of the BBTC are varied. Fetch engine and component areas for BBTC with trace table and BBTC with NTP are summarized in Tables VII and VIII.

3.3.2 *Instruction Cache Parameters.* In experiments that model an IC-only fetch engine, we use a two-way set associative, two-way interleaved instruction cache with 64 byte lines, and LRU replacement. For the *IC_HYB_BPRED* unrestricted component area experiments, the fetch engine area is comprised

Table VII. BBTC–TT Component Areas and Corresponding Fetch Engine Areas

Fetch Engine Area (KB)	Trace Table Area (KB)	Rename Table Area (KB)	Block Cache Area (KB)
268	32	8	128
436	64	16	256
772	128	32	512
1444	256	64	1024

Table VIII. BBTC-NTP Component Area and Corresponding Fetch Engine Areas.[a]

Fetch Engine Area (KB)	Rename Table Area (KB)	Block Cache Area (KB)	NTP Area (KB)
1746	8	128	1510
1930	16	256	1558
2250	32	512	1606
2842	64	1024	1654

[a]NTP area varies as size of block cache index/area varies and no trace table is included.

Table IX. Fastforward Numbers for Benchmarks[a]

Benchmark	Input	Fastforward (Insts)
164.gzip	ref graphic	77.3 B
176.gcc	ref expr	1.3 B
186.crafty	ref	72.8 B
197.parser	ref	183.8 B
252.eon	ref rushmeier	36.3 B
253.perlbmk	ref diffmail	13.3 B
255.vortex	ref lendian3	28.3 B

[a]Benchmarks are fast forwarded and then warmed up for 300 M instructions before statistics gathering.

of the area for the IC and the branch predictor, with the branch predictor area held constant. These parameters are listed in Table V. The IC areas and the area of the entire fetch engine are listed in Table VI.

3.4 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. Seven integer benchmarks (*gzip, gcc, crafty, parser, eon, perlbmk*, and *vortex*) and five floating-point benchmarks (*wupwise, mesa, art, facerec*, and *ammp*) were used in the experiments.

Our initial experiments demonstrated little performance benefit from larger fetch engines on the floating-point benchmarks. We suspect that this is because they have a small text size and are highly predictable. Thus, results for floating-point benchmarks are not shown and can be found in Co and Skadron [2003].

Simulations are fast-forwarded according to the numbers in Table IX [Sherwood et al. 2001], then run in full-detail cycle-accurate mode (without statistics-gathering) for 300 million instructions to train the caches—including the L2 cache—and the branch predictor before statistics gathering is started.

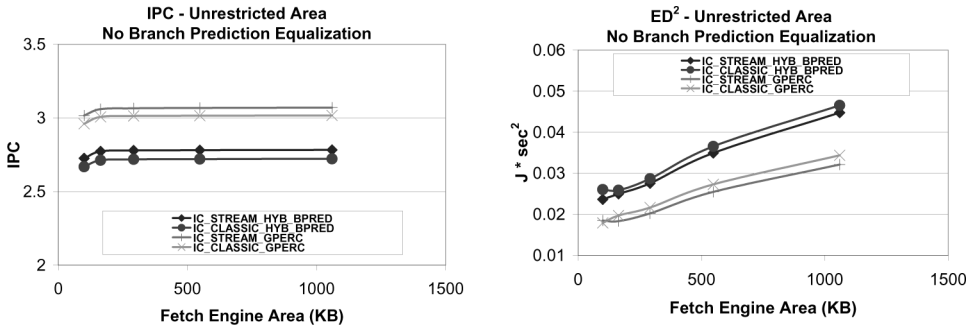


Fig. 3. IPC and ED^2–IC fetch engines using hybrid-branch predictor versus global perceptron branch predictor.

This interval was found to be sufficient to yield representative results [Haskins and Skadron 2003].

The individual results for each benchmark exhibited similar trends. Therefore, our results are presented as the average of the benchmarks.

4. FETCH ENGINE AREA EXPLORATION

We performed a comparison of the fetch engine designs listed in Table III. The areas listed in Tables VI, VII, and VIII were used. STC and BBTC associativity and area were varied and the IPC and energy-delay squared (ED^2) were analyzed. We chose to examine ED^2 as a metric because it considers both power dissipation and performance and is voltage independent. Increased associativity improved the IPC for CTC, STC, and BBTC, but showed only modest improvement in ED^2. We present direct-mapped results for these fetch engines, because it represents the worst-performing associativity and ED^2. Results for other associativities did not change the relationship between the fetch engines studied and so are not presented for the sake of space. The results for IC fetch engines all use a two-way associative, two-way interleaved IC.

4.1 Unrestricted Component Area

Each fetch engine was simulated using its published best branch predictor or trace predictor configurations. First, we considered the performance of two branch predictors in the evaluated fetch organizations: the hybrid branch predictor, which is used in the Alpha 21264, and the global perceptron branch predictor, which is considered the best branch prediction algorithm available today.

Figure 3 shows the IPC and ED^2 of the IC fetch engines using the different predictors. When comparing the *STREAM* models, we see that the fetch engine containing the global perceptron has a 10.3–10.7% better IPC. When comparing the *CLASSIC* models, the fetch engine with the global perceptron again has an average of 11% higher IPC. This improvement in IPC is as a result of the improved branch predictor accuracy of the global perceptron predictor. Both *CLASSIC* and *STREAM* global perceptron fetch models show a 26–28% better (lower) ED^2. Since the IC with global perceptron fetch engines far outperform

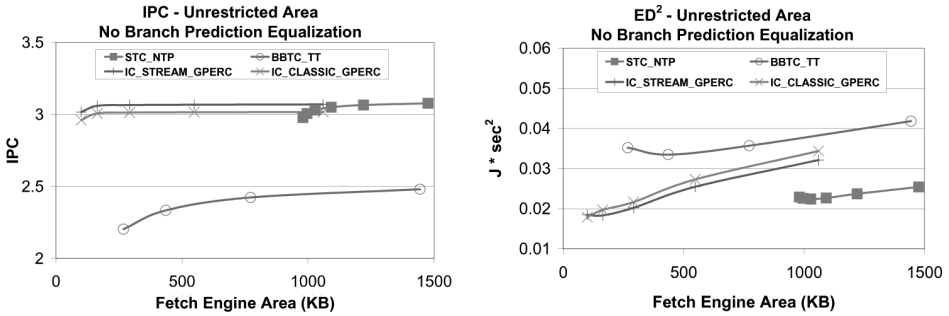


Fig. 4. IPC and ED^2–Unrestricted area fetch engines (STC, BBTC, and IC).

those with hybrid branch predictor, the remainder of our experimental results exclude fetch engines using a hybrid branch predictor.

Figure 4 shows the IPC and ED^2 of the initial fetch organizations evaluated. When comparing the *STC-NTP* to *IC_STREAM_GPERC*, *IC_CLASSIC_GPERC*, and *BBTC_TT*, it is clear that there is a difference in performance. Compared to *STC_NTP*, *IC_STREAM_GPERC*'s IPC is 0.24% worse, *IC_CLASSIC_GPERC* is 2.0% worse, and *BBTC_TT* is 19% worse. We believe that the *BBTC_TT*'s lower IPC is because of poor trace predictions provided by the trace table. Because of the *BBTC_TT*'s poor performance, for the remainder of this paper, we exclude *BBTC_TT* results. The *BBTC_TT*'s poor performance result does not conflict with Black's et al. work [1999], which uses perfect trace prediction. However, the BBTC is still included in the branch prediction isolation section.

The performance difference between STC_NTP and the other fetch engines could be the result of two factors: branch prediction accuracy and instruction storage order in the caches.

To understand the interplay between branch prediction accuracy and instruction storage order, we evaluate the additional fetch engine designs listed in Table IV. These fetch engines still contain either an IC, STC, or BBTC, but all use the same predictor (NTP) to make predictions. For the IC, the multiple-branch predictions provided by the NTP are consumed by the branch predictor one at a time. We chose to use the NTP to isolate branch prediction effects, because adapting it to work with the IC and BBTC was conceptually straightforward and would make comparisons to CTC and STC fetch engines simpler. The results of these experiments are discussed in Section 4.2.

4.1.1 *The Effects of Trace Construction Heuristics.* Rosner et al. [2003] presented a taxonomy and evaluation of heuristics for constructing long atomic traces for high instruction coverage. The trace construction heuristic chosen for a trace cache has a potentially significant effect on the trace cache hit rate, trace prediction accuracy, and overall performance and energy efficiency of the front end.

For the STC results evaluated in the other sections of this paper, the trace construction heuristic used has a maximum trace length (MTL) of 16, 6 maximum internal branches, and must terminate on indirect branches. We

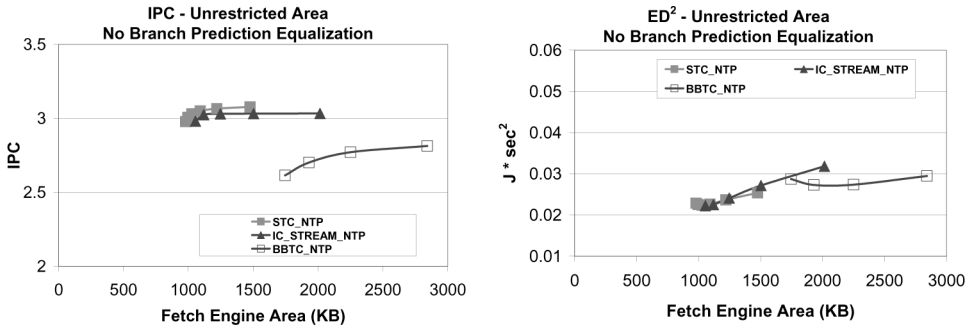


Fig. 5. Branch prediction tsolation (IPC and ED^2)—Unrestricted area fetch engines that use NTP.

evaluated several static trace construction heuristics on the STC-based fetch organization.

In our experiments, we explored terminating traces at backward branches, at calls, and at both backward branches and calls together. The base trace definition for comparison terminated traces only upon encountering an indirect branch for any MTL. Because of the nature of the trace identifier used in our STC model, we required that all traces must still terminate on indirect branches in addition to any of the other terminating conditions evaluated. This is reasonable since indirect branches are often difficult to predict.

For the SPECcpu2000 Integer benchmarks on an STC-based fetch design, we varied the maximum trace length (MTL) (16–256 instructions) while holding the associativity (four-way) and number of cache lines constant (8 K lines) to reduce cache conflict effects. The maximum number of branches allowed within each trace was also varied (4–64 branches).

Figure 6 shows the results for 16, 64, and 256 MTL, all normalized to results for 16 MTL traces, which terminate only on indirect branches (*16_base*). Compared with 16 MTL traces terminating only with indirect branches, additional terminating conditions have varying effects, depending on the benchmark. On average, varying these terminating conditions does not improve performance for the integer benchmarks, with the exception of *mcf*. As the trace length is increased, this effect is magnified, but again, IPC is not improved over the 16 MTL trace definition.

This result agrees with the results suggested by Rosner et al. [2003], which suggests that techniques that are able to use dynamic information (through the use of additional hardware to monitor dynamic behavior) yield better front-end performance. The design space for evaluating using dynamic information in trace construction is interesting and merits further exploration. However, this space is large and requires the addition of extra hardware. Adding hardware here would only increase the overhead that the trace cache fetch organizations would need to overcome to achieve equivalent or better energy efficiency than instruction cache-only fetch organizations. Therefore, we chose to limit our evaluation of trace construction heuristics to static trace construction heuristics, which require no additional hardware.

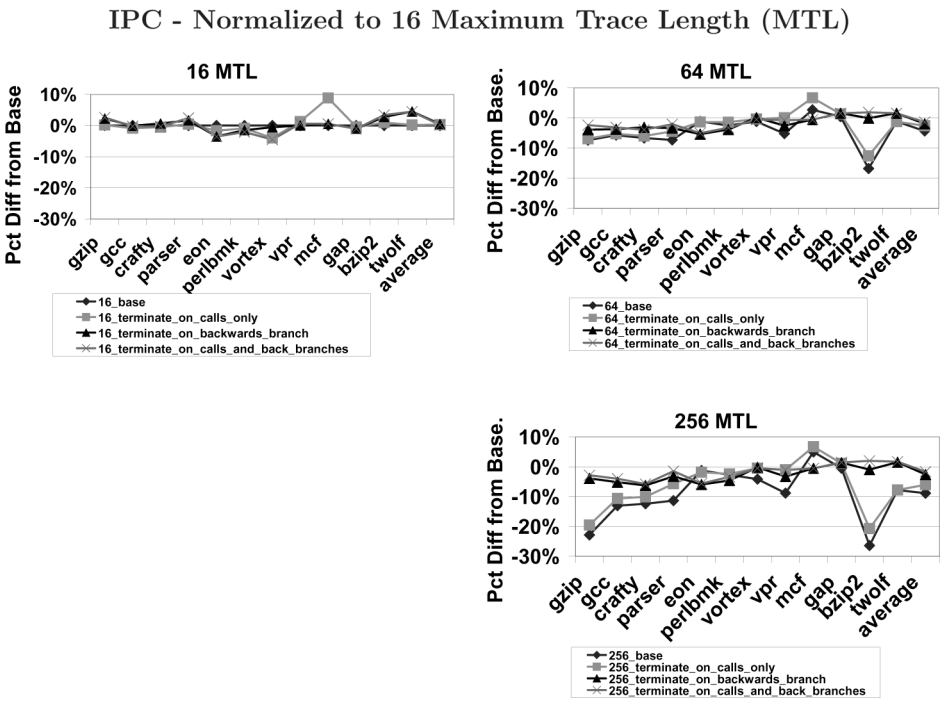


Fig. 6. IPC performance of different trace construction heuristics for different maximum trace lengths (MTL)—16, 64, and 256 instructions. Trace construction heuristics shown: base, terminate on backward branch, terminate on calls, and terminate on backward branches and calls. All results are normalized to 16 MTL base trace construction heuristic results, so only points above 0% show an improvement over base method.

4.2 Isolating Branch Prediction Effects

The different performance of the fetch engine designs in the previous section could be because of a combination of better branch prediction and more effective instruction storage order. To explore these two factors, we repeated the previous experiment with an additional set of fetch engine designs. Both IC and BBTC fetch engines were altered to include an NTP as the branch prediction mechanism.

One might think that a better approach to isolating branch prediction effects might be to probabilistically adjust the branch prediction accuracy of the respective branch/trace predictors in each fetch engine. The rationale for probabilistically altering branch predictions to make the prediction accuracy equal across fetch engines is to eliminate possible negative interactions between predictor and cache. However, probabilistic boosting of branch prediction accuracy assumes that all branches in a program are of equal weight and of equal predictability. This is not true. It is known that mispredictions are often clustered together. Therefore, we choose to isolate the effects of differing quality branch/trace predictors by supplying each type of cache with an NTP. We choose the NTP because the STC requires multiple branch predictions,

which are not easily provided by the branch predictors we use in our experiments. Our goal is to evaluate the differences in instruction storage order (IC, STC, and BBTC). We feel that using the NTP with each of the caches is a better way to isolate the effects of different branch prediction accuracy between predictors.

Augmenting the BBTC with an NTP improves the IPC for the BBTC. Compared to *STC_NTP*, the *BBTC_NTP*'s IPC is 8.6% worse compared to *BBTC_TT*'s 19% worse. *IC_STREAM_NTP*'s IPC is 1.43% worse than *STC_NTP*, compared to *IC_STREAM_GPERC*'s 1.95% worse. This demonstrates that branch prediction plays an important role in the performance of the fetch engines. Figure 5 shows the results.

Branch prediction accuracy is the dominant factor affecting performance and energy efficiency. This is demonstrated through our evaluation of the fetch engines with the NTP. The only difference in the *BBTC_TT* fetch organization and the *BBTC_NTP* fetch organization is the trace prediction mechanism. The trace table makes poorer predictions than the NTP. Substituting the more accurate NTP for the TT attains a 13% IPC improvement and a 29% improvement in ED^2. This result, combined with the similar performance of *STC_NTP* and *IC_STREAM_NTP* in Figure 5, demonstrate that branch prediction accuracy is much more important than instruction storage order. To make a fair comparison of fetch engine designs, equivalent branch predictors must also be used. Otherwise, our results show that what will really be compared is the differences in branch prediction accuracy of the branch predictors.

4.3 Restricted Component Area

Clock rates in modern processors are increasing rapidly, while wire latencies are not scaling accordingly. As a result, the time delay to access structures is becoming a serious obstacle. To account for access time considerations, we performed an experiment where the area of each component of each fetch engine was limited to fixed-area budgets ranging from 2 to 512 KB. We perform this experiment with all fetch engines, using direct-mapped structures, where applicable. For example, for BBTC simulations, the area of each of the BBTC components (trace table, rename table, block cache), backing instruction cache, and branch predictor was limited to a fixed area.

4.3.1 *Performance Evaluation.* Figures 7 and 8 show that at 32 KB and larger component areas, the *STC_NTP* achieves up to 5% higher IPC than *IC_CLASSIC_GPERC*. The *STC_NTP* does not do as well at component areas smaller than 32 KB, because of the NTP's poor performance with tight area limitations. The *IC_NTP* fetch organizations do not start to outperform *IC_CLASSIC_GPERC* until much higher component areas. The effect of increasing leakage is discussed in the following section.

4.3.2 *Leakage Effects.* Total power dissipation as a result of chip leakage is projected to exceed total dynamic power as feature sizes reach 65 nm [SIA 2001]. To ensure that we consider the energy efficiency of CTCs, STCs, and ICs both now, and in future process technologies, we examined the results of

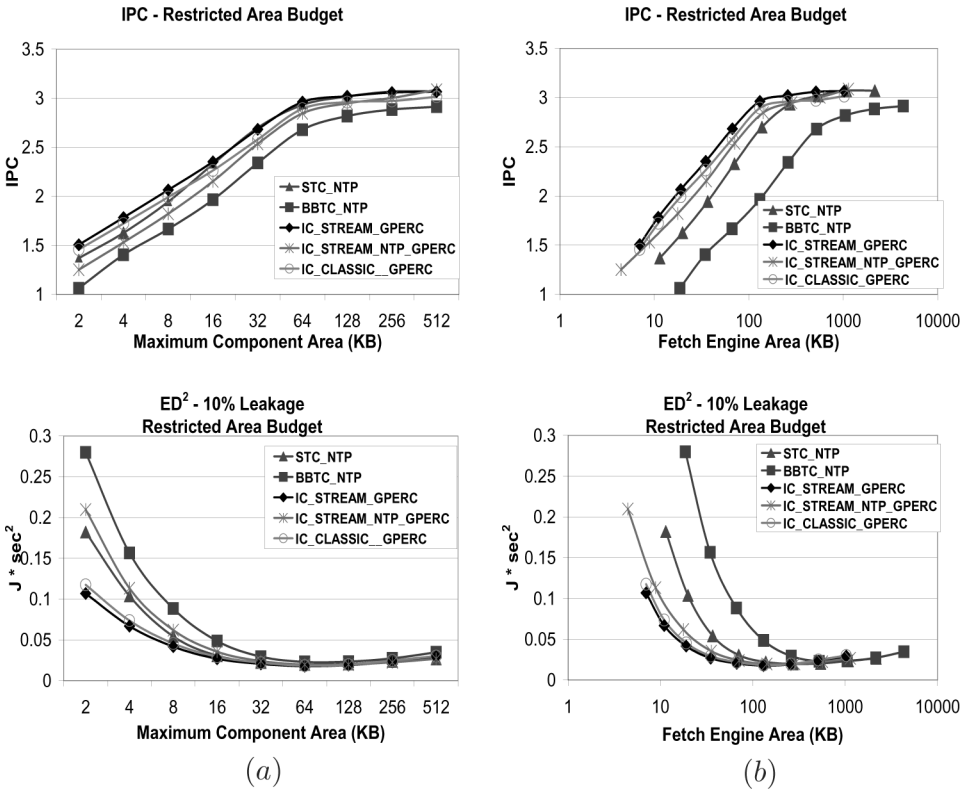


Fig. 7. IPC and ED^2 for equal area fetch components: (a) by maximum component area and (b) by total fetch engine area.

varying the leakage ratio from 10 to 50% of maximum power dissipation. Since leakage effects did not affect the IPC in our results, we do not present IPC results for other leakage ratios in this section.

We found that when component areas are restricted to account for access latency, and a uniform temperature is assumed, increased leakage ratio has little effect on the relative energy efficiency of the evaluated fetch engine designs. In other words, the energy efficiency relationship between the fetch designs does not change. The increasing leakage ratio acts only as a vertical offset. This is because of the fact that the components for the respective fetch organizations are relatively equal in size as a result of the area restriction and, since all these structures are SRAM arrays, leakage just tracks area.

However, leakage is dependent on temperature, which, turn is dependent on the architecture's floorplan. To estimate these thermal effects, we calculated the power density for the main fetch engine structures (IC, branch predictors, STC, and NTP) using a beta version of Cacti 4.0 [Shivakumar and Jouppi 2001] that has been updated compared to version 3.0 to include both subthreshold and gate leakage, various ways to model SRAM, and modeling of different power modes. A process technology of 0.13 μm is assumed. Results are shown in Figure 9. For each fetch design, the power density calculations include only the instruction

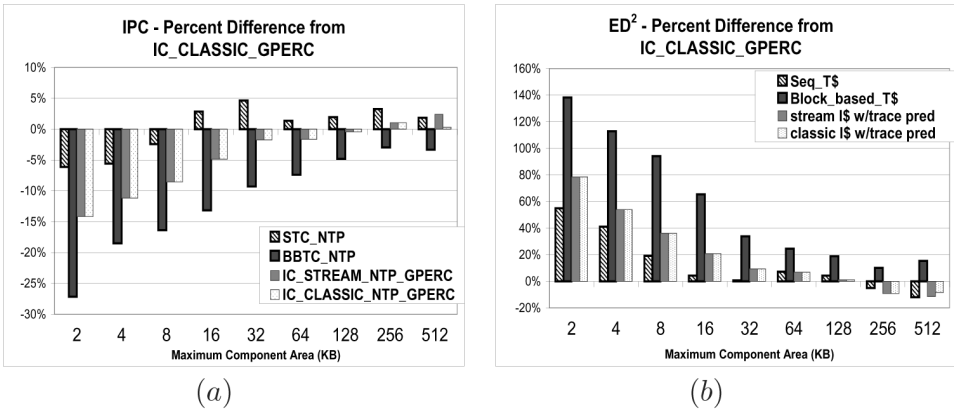


Fig. 8. Percentage difference of (a) IPC and (b) ED^2 relative to *IC_CLASSIC_GPERC*.

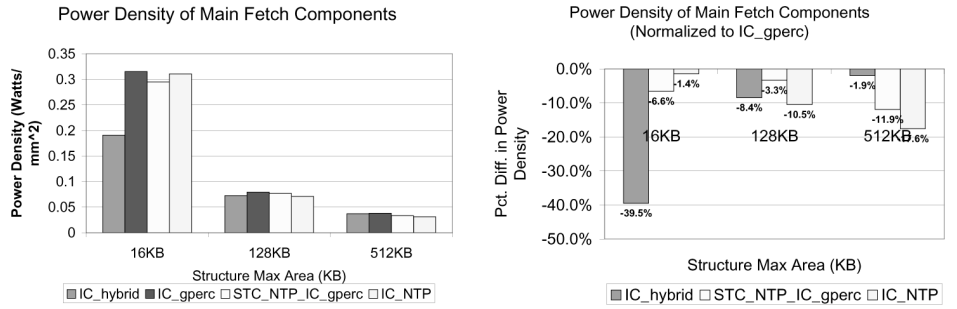


Fig. 9. Power density for fetch organizations evaluated by maximum component area.

or trace cache (or both) and the prediction units. The results shown are for the 16, 128, and 512 KB component areas.

We found that for the component areas studied, the power densities of trace-cache-based fetch engines were not more than that of a fetch unit containing an instruction cache and global perceptron branch predictor. In fact, Figure 9 shows that the power density of the STC fetch design is always slightly lower than IC with global perceptron. The STC power density is lower than that of the IC because the STC's instruction storage layout provides more useful instructions per fetch bundle than ICs. Basically, on average, the STC will require fewer accesses due to more efficient storage, which, in turn, results in fewer total accesses. This indicates that the leakage ratio for STC-based fetch designs will not increase at a higher rate than IC-based fetch designs because of temperature effects. (Since all other hardware is held constant in the study, we did not consider other components that might contribute to temperature.)

While estimating power density is a simplistic approach to evaluating the thermal effects on leakage rate, it is a good indicator and a more detailed evaluation including floor planning and thermal management are outside of the scope of this work. We assume that, for each size, the STC or IC components would be placed similarly, so that neighboring functional units temperatures would have similar effects.

STC gets higher IPC than the other fetch engines starting at 32 KB component areas, followed closely by *IC_STREAM_GPERC*, *IC_CLASSIC_GPERC*. However, the ED^2 for component areas larger than 16 KB is approximately the same for the STC and IC_GPERC fetch engines. Higher IPC is offset by more structures to access and more power at a given area.

When the components of each fetch engine are restricted to a specific area budget, the effect of leakage is not significant, compared to the results when fetch engine area budget is unrestricted. Each component in each fetch engine is no larger than a fixed area, so naturally the effect of leakage for each fetch-engine becomes similar.

For the BBTC, the published best configuration has an 8-k entry trace table (128 KB) and 4-k entry block cache (512 KB). The block cache (the main component of the BBTC) may be penalized under the area restriction. This explains why the BBTC does not fare as well for the restricted component area experiments.

5. AHEAD PIPELINED NTP

Faster clock rates lead to shorter cycle times. Shorter cycle times makes accessing larger structures more challenging. Our experimental results show that branch prediction accuracy is an important factor in fetch engine performance. Our results show that the NTP provides better branch prediction accuracy than a hybrid branch predictor and similar branch prediction accuracy. However, at very small component areas, the NTP performs poorly. Ahead pipelining is one technique to enable a structure to be larger and still be able to produce output each cycle.

Ahead pipelining initiates a prediction many cycles in advance of when it is needed to hide access latencies. To do so, older information must be used to generate a set of the predictions for selection. At the last moment, the most current information is used to select the final prediction. Patt et al. [1993] evaluates pipelined access to a branch address cache to perform multiple-branch prediction. Jimenez [2003], Seznec [2004], Seznec et al. [1996], and Tarjan and Skadron [2004], evaluate ahead-pipelining single-branch predictors in order to obtain better prediction accuracy (enable larger-branch predictor structures), while considering the impact of delay. We apply the ahead-pipelining concept to the NTP trace prediction mechanism and perform a set of experiments in which we compare the performance and energy efficiency of fetch engines with the pipelined NTP.

Ahead pipelining the NTP is a way to reduce some of the performance loss from reduced area because of cycle-time restrictions. It allows the planning of structures larger than can be accessed within a single cycle and yet still produce accurate output each cycle.

Roughly 1–2 KB can be accessed within a single cycle [Jimenez et al. 2000]. If a structure can be pipelined to two stages, the structure could be made as large as 4–8 KB. As the depth of the pipeline increases, the area for a particular structure that can be accessed roughly quadruples [Jimenez 2003].

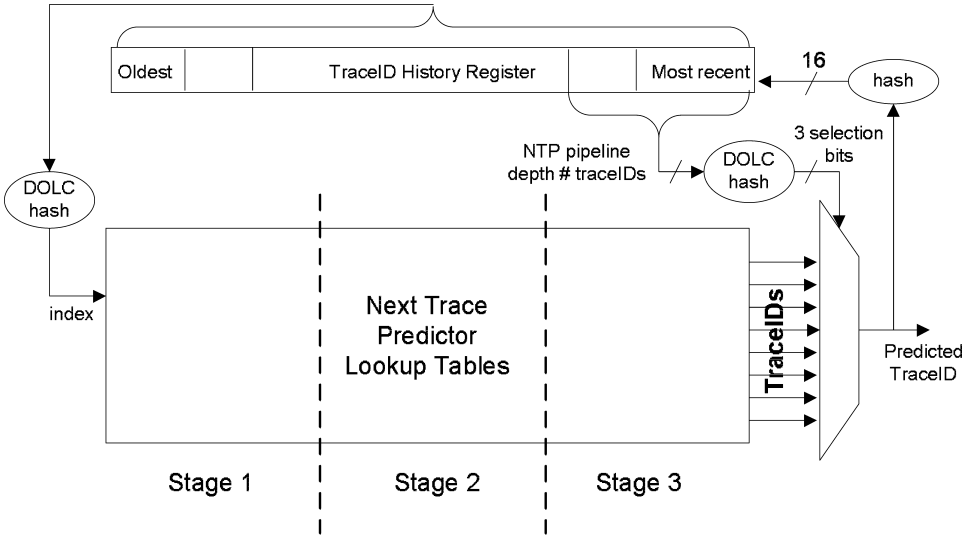


Fig. 10. Sample three-stage ahead-pipelined NTP.

Figure 10 shows an example of a three-stage ahead-pipelined NTP. Several cycles before a prediction is needed, the current trace history is used to select a range of entries to extract from the NTP tables. Since several trace predictions are in-flight in the NTP pipeline, this history does not include information about the immediately preceding trace. To capture information about immediately preceding traces, the range of entries is reduced to a single entry in the cycle immediately before it is needed. The most recent history is hashed to select which of the range of entries is used. This technique is similar to the technique used by Patt et al. [1993] and Jimenez [2003], except that we apply it to next-trace prediction, which is a form of implicit multiple-branch prediction. Trace misprediction latency is modeled as the depth of the NTP pipelining (flushing the NTP pipeline). However, since the backing-branch predictor is used in a misprediction, much of the latency as a result of the NTP ahead pipelining is used to fetch via the slow path mechanism.

We compare 1–512 KB NTP component areas, assuming that the NTP pipeline depth must increase as the area is increased. We vary the pipeline depth from 1 (unpipelined) to 5 and vary the range of entries chosen by the incomplete history from 1 to 8 (see Table X).

Figure 11 shows the NTP correct trace prediction accuracy of the NTP for varying pipeline depths and selection ranges for the 16-KB component area. (All structures in the NTP can be no larger than 16 KB). We choose to show the NTP trace prediction accuracy in order to demonstrate the potential for pipelining the NTP. The bar labeled *Depth_1* represents the maximum attainable NTP prediction accuracy since it is the single-cycle trace prediction accuracy. As the depth of the pipeline increases, if only a single entry is selected in the early stages, the trace prediction accuracy rapidly decreases. This is because

Table X. NTP Component Areas and Corresponding NTP Pipeline Depth[a]

Component Area (KB)	Pipeline Depth
1	1
2	1
4	2
8	2
16	3
32	3
64	4
128	4
256	5
512	5

[a]Number of entries selected in advance is varied at values 1, 2, 4, and 8.

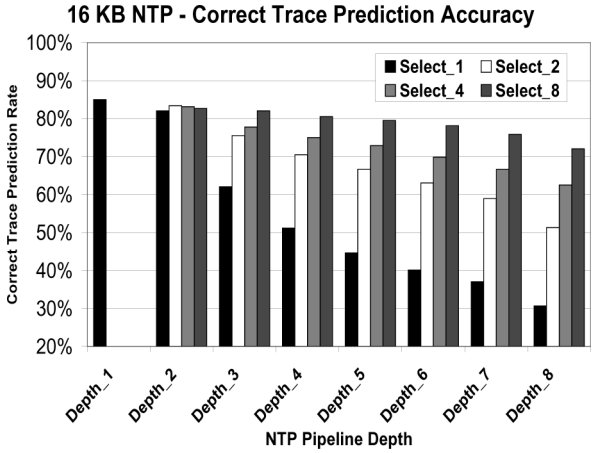


Fig. 11. NTP correct trace prediction accuracy as affected by pipeline depth and number of entries selected for final prediction.

of the use of only older history and no newer history to make the next-trace prediction. Increasing the selection range of entries at prediction time improves the prediction accuracy. As the number of entries selected is increased to 8, the difference in trace prediction accuracy from the original, nonpipelined NTP rapidly decreases. As the depth of the pipeline increases, the trace prediction declines because of the use of more old history and less new history to make the trace prediction.

Figure 12 compares the NTP trace prediction accuracy of comparable pipelined NTP design points. A nonpipelined 1-KB area, NTP can be compared to a progressively larger, more deeply ahead-pipelined NTP. The dark colored section of the bar represents the trace prediction accuracy when ahead pipelined, while the shaded area of the bar represents the trace prediction accuracy if the same area structure were to be accessible within a single cycle. These results show that the trace prediction accuracy does not suffer too much of a penalty from being pipelined.

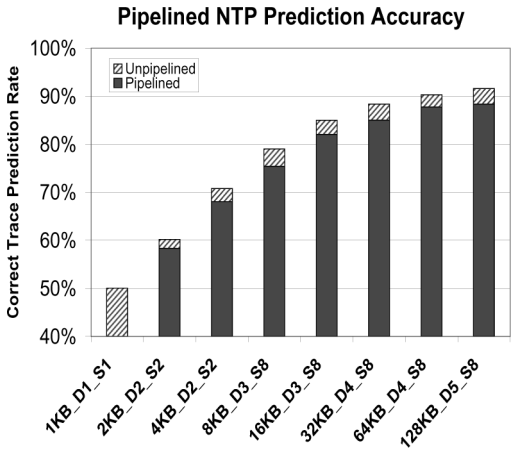
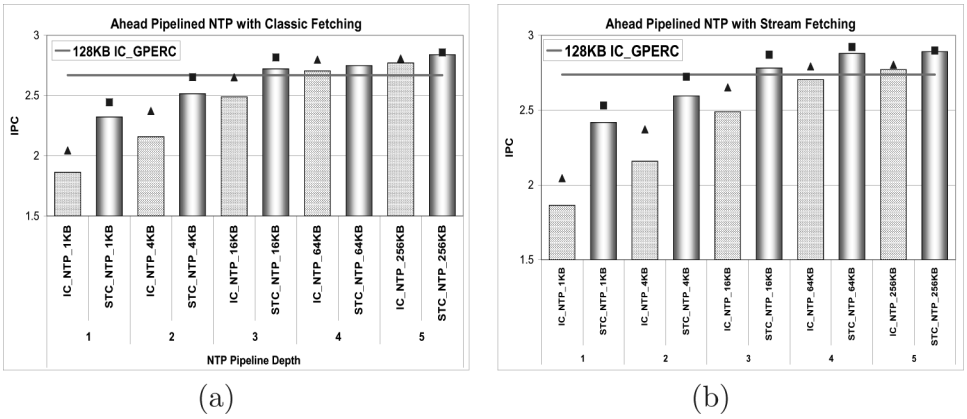


Fig. 12. Improvement of NTP correct trace prediction accuracy as NTP table areas increase. Labels are of the form X_Y_Z, where X = NTP Area, Y = NTP Pipeline Depth, and Z = Number of entries selected for final trace prediction.



(a) (b)

Fig. 13. Classic (a) vs. stream fetching (b). IPC of 128 KB IC and STC fetch engines with ahead-pipelined NTP relative to 128 KB IC with global perceptron predictor. (Triangles and squares represent IPC when area of NTP is doubled for IC or STC fetch organizations.)

Figure 8 shows that a 32-KB STC, accessible in a single cycle, has similar ED^2 to *IC_CLASSIC_GPERC*; 32 KB is the maximum area accessible using a three-deep pipelined NTP. Since 32 KB is not accessible in a single cycle, we compare the performance of our 32 KB three-deep pipelined *STC_NTP*, which can provide one prediction per cycle to a 2-KB nonpipelined *STC_NTP*, which can be accessed in a single cycle. We find that a 17.2% improvement in IPC and 29% reduction in ED^2 is observed. (See Figure 13.) This suggests that with ahead pipelining, STCs can provide significant performance benefits and increased energy efficiency.

Figure 13 shows the IPC of IC and STC fetch engines using an ahead-pipelined NTP. These results are compared to the IPC of a 128 KB IC with a global perceptron predictor (*IC_GPERC*, the global perceptron branch predictor

simulated is accessible within a single cycle). Large STC and IC (128 KB) are used to minimize the effect of cache misses in the experiment. The horizontal line represents the IPC of the 128 KB *IC_GPERC*. Each bar in the graph represents a fetch engine with the given area NTP. The symbol above each bar (triangles for IC, squares for STC) represents the IPC result when doubling the respective area of the NTP, but keeping the same NTP pipeline depth. The doubled area of the NTP represents the maximum NTP area that may be reached without needing to add an additional pipeline stage. The graphs show that ahead pipelining the NTP dramatically increases the performance of both the IC and STC fetch engines up to 16 KB (NTP pipeline depth 3). At 16 KB, the STC fetch engine with *CLASSIC* fetching gets 2–5.5% higher IPC than the *IC_GPERC* and with *STREAM* fetching gets 1.5–5% higher IPC. The *IC_NTP* configuration does not outperform the *IC_GPERC* design until the NTP is expanded to 64 KB (1.3–4.8% higher IPC). This IPC gain is about the same as the *STC_NTP* at 16–32 KB. We believe that the lower IPC for the IC compared to the STC with NTPs of 16 KB and smaller is as a result of instruction storage order differences. At lower areas, the NTP has greatly reduced prediction accuracy, which could be partially masked by the STC's instruction storage order (traces). We believe that the IC's storage of instructions in static program order is more sensitive to this drastically poorer misprediction rate.

The graphs also show that increasing the NTP area and NTP pipeline depth continue to improve the IPC, but at a diminishing rate than from 1 to 32 KB. Empirically, an NTP pipeline depth of 3 exhibits reasonable performance for any of the fetch engines using *CLASSIC* fetching. As the pipeline depth increases beyond this point, we see diminishing returns in the performance gain. We believe this decrease in IPC is because of the fact that older history is used to make the initial region selection from the NTP. With *STREAM* fetching, an NTP pipeline depth of 4 still achieves some performance improvement.

6. CONCLUSIONS AND FUTURE WORK

We evaluate the energy efficiency of trace caches compared to instruction caches. Our experiments show that when fetch components are not constrained by realistic factors, such as access time and resource constraints, fetch engines based on STCs are more energy efficient, while providing a significant performance improvement over IC-only fetch engine organizations. With these idealistic assumptions, the ED^2 results show that although an STC and its supporting components may occupy more chip area than an IC-only fetch engine, the STC yields better energy efficiency overall because of better opportunities for accessing smaller area fetch engine components and minimizing energy expenditures via clock-gating techniques. These results represent the theoretical benefit of trace caches, which stem primarily from the implicit multiple-branch prediction of the NTP, but also from the benefit of storing instructions as traces as opposed to static program blocks.

To further understand the source of the STC's better energy efficiency when no restrictions on resources and access delay are imposed, the cache designs are evaluated when their branch predictors are normalized to use the NTP. We

find that branch prediction accuracy plays a critical role in the performance of all of the fetch engines. For BBTC, we find that replacing the TT with the NTP achieves a 13% improvement in IPC and 29% improvement in ED^2. This result emphasizes the continuing importance of research to improve branch prediction accuracy.

However, when considering the impact of access delay by limiting the area of the fetch engine structures, we find that the benefit of STC fetch engines compared to IC fetch engines is more modest than when delay is not considered. This is because the NTP and STC are not as effective at smaller areas. When component area is limited to 32 KB or less, IC fetch organizations have much better IPC and ED^2. However, at larger component areas, the better branch prediction capability of the NTP and more efficient instruction delivery of the STC attain only modestly better IPC and ED^2 than the IC fetch organizations.

To address the effect of increasing access delay on STCs, we introduce and evaluate an ahead-pipelined NTP. When comparing to a single-cycle accessible, non-pipelined, 2-KB *STC_NTP*, a two-deep pipelined 32-KB *STC_NTP* can provide a 17.2% IPC improvement and 28.9% ED^2 improvement. Our experiments show that ahead-pipelining the NTP to four stages (allowing 64 KB of area) allows the *IC_CLASSIC_NTP* to get 1.3–4.8% higher IPC than an IC, with 4-KB global perceptron predictor. The *STC_NTP* fetch engine achieves similar IPC improvement at an NTP pipeline depth of 3.

Future directions for this work include exploring the energy efficiency of other fetch engine designs and incorporating the dynamic prediction-directed trace cache [Hu et al. 2003] and filter trace-cache [Tang et al. 2001] models. Our results also suggest that the large design space for trace-construction and trace-prediction techniques is a fertile area for future work.

ACKNOWLEDGMENTS

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

REFERENCES

BAHAR, R. I., ALBERA, G., AND MANNE, S. 1998. Power and performance tradeoffs using various caching strategies. In *Proceedings of the 1998 International Symposium on Low Power Electronics and Design*. ACM Press, New York. 64–69.

BLACK, B., RYCHLIK, B., AND SHEN, J. 1999. The block-based trace cache. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*. IEEE Computer Society Press, Washington, DC. 196–207.

BOSE, P., BROOKS, D., BUYUKTOSUNOGLU, A., COOK, P., DAS, K., EMMA, P., GSCHWIND, M., JACOBSON, H., KARKHANIS, T., KUDVA, P., SCHUSTER, S., SMITH, J., SRINIVASAN, V., ZYUBAN, V., ALBONESI, D., AND DWARKADAS, S. 2002. Early-stage definition of lpx: A low power issue-execute processor. In *Proceedings of the Workshop on Power-Aware Computer Systems held in conjunction with HPCA-8*. Cambridge, MA.

BROOKS, D., TIWARI, V., AND MARTONOSI, M. 2000. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*. 83–94.

BURGER, D. C. AND AUSTIN, T. M. 1997. The SimpleScalar tool set, version 2.0. *Computer Architecture News 25*, 3 (June), 13–25.

Co, M. and Skadron, K. 2003. Evaluating the energy efficiency of trace caches. Tech. Rep. CS-2003-19, University of Virginia, Department of Computer Science.

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

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

Fahs, B., Bose, S., Crum, M., Slechta, B., Spadini, F., Tung, T., Patel, S. J., and Lumetta, S. S. 2001. Performance characterization of a hardware mechanism for dynamic optimization. In *MICRO34*. IEEE Computer Society, Washington, DC. 16–27.

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

Haskins, J. W., Jr. and Skadron, K. 2003. Memory reference reuse latency: Accelerated sampled microarchitecture simulation. In *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software*. 195–203.

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

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

ITRS 2001. *International Technology Roadmap for Semiconductors*. http://www.itrs.net/Links/2001ITRS/Home.htm.

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

Jimenez, D. 2003. Reconsidering complex branch predictors. In *In Proceedings of the 9th International Symposium on High Performance Computer Architecture*. 43–52.

Jimenez, D. A., Keckler, S. W., and Lin, C. 2000. The impact of delay on the design of branch predictors. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*. ACM Press, New York. 67–76.

Johnson, J. 1994. Expansion caches for superscalar processors. Tech. Rep. CSL-TR-94-630, Computer Science Laboratory, Stanford University.

Kim, N., Flautner, K., Blaauw, D., and Mudge, T. 2002. 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*. IEEE Computer Society Press, Washington, DC. 219–230.

Oberoi, P. and Sohi, G. 2003. Parallelism in the front-end. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*. ACM Press, New York. 230–240.

Parikh, D., Skadron, K., Zhang, Y., Barcella, M., and Stan, M. 2002. Power issues related to branch prediction. In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture*. 233–244.

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

Peleg, A. and Weiser, U. 1995. Dynamic flow instruction cache memory organized around trace segments independent of virtual address line. US. Patent Number 5, 381, 533.

Ramirez, A., Santana, O., Larriba-Pey, J., and Valero, M. 2002. Fetching instruction streams. In *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society Press, Washington, DC. 371–382.

Reinman, G., Austin, T., and Calder, B. 1999. A scalable front-end architecture for fast instruction delivery. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*. IEEE Computer Society. 234–245.

ROSNER, R., MOFFIE, M., SAZEIDES, Y., AND RONEN, R. 2003. Selecting long atomic traces for high coverage. In *Proceedings of the 2002 International Conference on Supercomputing*. ACM Press, New York, NY. 2–11.

ROSNER, R., ALMOG, Y., MOFFIE, M., SCHWARTZ, N., AND MENDELSON, A. 2004. Power awareness through selective dynamically optimized traces. In *Proceedings of the 31th Annual International Symposium on Computer Architecture*. IEEE Computer Society, Washington, DC. 162.

ROTENBERG, E., BENNETT, S., AND SMITH, J. 1996a. Trace cache: A low latency approach to high bandwidth instruction fetching. Tech. Rep. 1310, Cs Dept. University of Wisconsin, Madison.

ROTENBERG, E., BENNETT, S., AND SMITH, J. 1996b. Trace cache: A low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*. 24–35.

ROTENBERG, E., JACOBSON, Q., SAZEIDES, Y., AND SMITH, J. 1997. Trace processors. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*. 138–148.

ROTENBERG, E., BENNETT, S., AND SMITH, J. E. 1999. A trace cache microarchitecture and evaluation. *IEEE Transactions on Computers 48*, 2, 111–120.

SEZNEC, A. 2004. Revisiting the Perceptron Predictor. Tech. Rep. 1620, IRISA.

SEZNEC, A., JOURDAN, S., SAINRAT, P., AND MICHAUD, P. 1996. Multiple block ahead branch predictors. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*.

SHERWOOD, T., PERELMAN, E., AND CALDER, B. 2001. 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*.

SHIVAKUMAR, P. AND JOUPPI, N. P. 2001. Cacti 3.0: An integrated cache timing, power, and area model. Tech. Rep. WRL-TR-2001/2.

SKADRON, K., STAN, M. R., HUANG, W., VELUSAMY, S., SANKARANARAYANAN, K., AND TARJAN, D. 2003. Temperature-aware microarchitecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*. 2–13.

SOLOMON, B., MENDELSON, A., ORENSTEIN, D., ALMOG, Y., AND RONEN, R. 2001. Micro-operation cache: A power aware frontend for the variable instruction length isa. In *Proceedings of the 2001 International Symposium on Low Power Electronics and Design*. ACM Press, New York. 4–9.

TANG, W., GUPTA, R., AND NICOLAU, A. 2001. Design of a predictive filter cache for energy savings in high performance processor architectures. In *Proceedings of the 2001 International Conference on Computer Design*. 68–73.

TARJAN, D. AND SKADRON, K. 2004. Revisiting the perceptron predictor again. Tech. Rep. CS-2004-28, University of Virginia, Department of Computer Science.

TARJAN, D., SKADRON, K., AND STAN, M. 2004. An ahead pipelined alloyed perceptron with single cycle access time. In *Proceedings of the 5th Workshop on Complexity-Effective Design*.

YEH, T.-Y. AND PATT, Y. N. 1992. A comprehensive instruction fetch mechanism for a processor supporting speculative execution. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*. IEEE Computer Society Press, Washington, DC. 129–139.

ZHANG, Y. AND YANG, J. 2003. Low cost instruction cache designs for tag comparison elimination. In *Proceedings of the 2003 International Symposium on Low Power Electronics and Design*. ACM Press, New York. 266–269.

ZYUBAN, V. AND STRENSKI, P. 2002. Unified methodology for resolving power-performance tradeoffs at the microarchitectural and circuit levels. In *Proceedings of the 2002 International Symposium on Low Power Eelctronics and Design*. ACM Press, New York. 166–171.

Received December 2004; revised February 2006; accepted June 2006