Energy-Aware Fetch Mechanism: Trace Cache and BTB Customization

Daniel Chaver, Miguel A. Rojas, Luis Pinuel, Manuel Prieto, Francisco Tirado
Dpto. Arquitectura de Computadores
Universidad Complutense, Madrid, Spain
{dani02, miguel.rojas, lpinuel, mpmatias, ptirado}@dacya.ucm.es

Michael C. Huang
Dept. of Electrical and Computer Engineering
University of Rochester
Rochester, NY, USA
{michael.huang}@ece.rochester.edu

ABSTRACT
A highly-efficient fetch unit is essential not only to obtain good performance but also to achieve energy efficiency. However, existing designs are inflexible and depending on program behavior, can be either insufficient or an overkill. We introduce a phase-based adaptive fetch mechanism that can be dynamically adjusted based on feedback information of the program behavior. This design adds very little hardware complexity and relegates complex tasks to the software components. It is also very effective: saving 26.8% and 34.1% fetch energy on average compared with a conventional and a trace cache-based fetch unit, respectively. At the same time, performance is improved by 5.7% and 0.6%, respectively.

Categories and Subject Descriptors
C.1.3 [**Processor Architectures**]: Other Architecture Styles-*Adaptable Architectures*.

General Terms
Design, Experimentation, Performance.

Keywords
Adaptive, Instruction Fetch, Profiling.

1. INTRODUCTION
Modern high-end processors rely on sophisticated branch prediction and instruction fetch mechanisms to achieve high performance *and* energy efficiency. Without a constant, smooth supply of instructions, the rest of the pipeline will not only perform poorly, but also use energy inefficiently. However, such sophisticated mechanisms are not without cost and can account for a significant fraction of energy consumption (*e.g.*, 22% in Pentium Pro).

A conventional fetch mechanism consists of an instruction cache and a branch predictor (including a direction predictor and a branch target buffer). The most basic instruction fetch mechanism can only supply a consecutive chunk of instructions from a single cache line. If a branch is predicted to be taken, the fetching of target instructions is delayed to the next cycle. This is referred to as SEQ1 [1]. A slightly more aggressive variation of this conventional fetch mechanism is to add the capability to fetch across multiple branches. In such an implementation, a multi-branch predictor is needed to provide *n* predictions per cycle. For example, a SEQ3 scheme [1] can fetch across up to 3 branches, provided all the targets remain in the same cache line. We note that the additional complexity of SEQ3 over SEQ1 is rather low: besides requiring a multi-branch predictor, the hardware only needs to "collapse" the fetched cache line to remove instructions between the taken branches and their respective targets.

Rotenberg *et al.* introduced the trace cache as a promising solution to obtain a high bandwidth instruction fetching with a very low latency [1]. The idea is to capture dynamic instruction sequences in an additional cache that stores *traces*. A trace is a sequence of at most *n* instructions and at most *m* basic blocks, identified by its starting address and *m-1* branch outcomes. Note that a multi-branch predictor is needed to provide *m-1* predictions per cycle. In a conventional trace cache design (referred to as CTC [2, 3]), the trace cache, the multi-branch predictor, and the I-cache are accessed simultaneously to reduce miss penalty when the trace cache misses. In a Sequential Trace Cache (STC) design [2, 3], instruction fetch is carried out in two phases. The trace cache and branch predictor are accessed first, and the I-cache is accessed later, only upon a trace cache miss. At the cost of longer latency when the trace cache misses, this approach can reduce the energy consumption due to unnecessary access of the I-cache when the trace cache hits.

Given all these different fetch options, the best strategy depends on the behavior of the program. Intuitively, the best option is the one that balances the front-end of the machine, which fetches the instructions, and the execution engine, which processes them. If the front-end can not keep up, the execution will take longer than necessary and this longer execution increases energy consumption due to clock distribution and other overhead. Conversely, an overly aggressive front-end can not further improve performance, and thus its high energy expenditure becomes unnecessary. Unfortunately, program behavior varies not only across applications but also within a single application. Hence, there is

* This work is supported in part by the Spanish Government grant TIC 2002-750, the *HiPEAC* European Network of Excellence, and the National Science Foundation through the grant CNS-0509270. Manuel Prieto was also supported by the Spanish Government MECD mobility program PR2003-0306.

ISLPED'05, August 8–10, 2005, San Diego, California, USA.

no fixed configuration that always works efficiently. An efficient system will be adaptive, selecting the fetch policy that works best for the code currently executing. Additionally, the size of various structures, such as the trace cache and the branch target buffer (BTB), also adjusts according to program demand.,

In this work, our focus is to design such a flexible fetch mechanism that can reduce energy consumption and improve performance without significantly increasing the design complexity. We adopt the general principle of on-demand resource allocation and propose a design that we call *Phase-Based Adaptive Fetch Mechanism* (PBAFM). In this scheme, the instruction fetch unit is adjusted periodically, at the boundary of *phases* to adapt to the changing program behavior. A phase is a period of execution with predictable behavior. In this paper, we use a simple strategy to identify program phases. We follow prior work [6, 7] and statically divide an application into *modules*. The dynamic execution of a static code module is a natural candidate of a phase: prior execution of a code module can be used to accurately predict behavior of future instances. Given the command to adapt, the hardware re-orchestrates the fetch mechanism using a different combination of the component structures. Furthermore, using existing circuit techniques [4, 5], the size of these structures is also adjusted according to the software command.

To simplify the design and reduce runtime overhead, we use a feedback-based approach relying on software components to identify resource demand and make a decision on the choice of configuration. The hardware, on the other hand, only provides the primitives to carry out the reconfiguration. We show that our proposal is straightforward to implement and is highly effective: it not only saves fetch energy by 26.8% and 34.1% compared to a conventional fetch unit (SEQ1) and a trace cache-based fetch unit respectively, it also improves the performance of execution.

The rest of this paper is organized as follows. We first discuss the rationale of our work in Section 2. In Section 3 we explain our adaptive fetch design. Section 4 describes the experimental framework. Evaluation results are shown in Section 5. Section 6 discusses related work. Finally, Section 7 presents some conclusions.

2. RATIONALE

Intuitively, what fetch mechanism works well depends on the application's characteristics. When the application exhibits high trace cache hit ratios, an STC will avoid the unnecessary waste accessing the I-cache in parallel and is thus more efficient than the CTC. On the other hand, when the application exhibits high trace cache miss ratios, the CTC may be a better mechanism since the STC incurs extra latency every time that the trace cache misses.

Given the wide variety of general-purpose applications, it is not surprising that there is no single optimal fetch mechanism or configuration. Figure 1 illustrates this point quantitatively. We simulate the execution of several SPEC CPU 2000 applications (the details of the experimental methodology will be described later in Section 4) and show the energy-delay product of the execution under different fetch policies and configurations. Figure 1-(a) shows the energy-delay product of a system using SEQ3 and CTC fetch policy with different trace cache sizes. We present normalized results using the energy-delay product of a baseline system with a SEQ1 policy. In Figure 1-(b) we vary the BTB size from 256 entries to 4096 entries and normalize the result to that of a processor with a smaller BTB (128 entries).

From these results, we can make several observations. First, different fetch mechanisms work differently depending on the application. For example, while SEQ3 works well for *parser* and *twolf*, CTC is a better choice for *gap* and *vortex*. Moreover, the optimal policy may depend on the specific configuration. For example, with a trace cache smaller than 64KB, CTC is the most efficient mechanism for application *gcc*, however, a bigger trace cache makes it less efficient than SEQ3. Finally, configuration details such as BTB size can also drastically affect the efficiency of program execution.

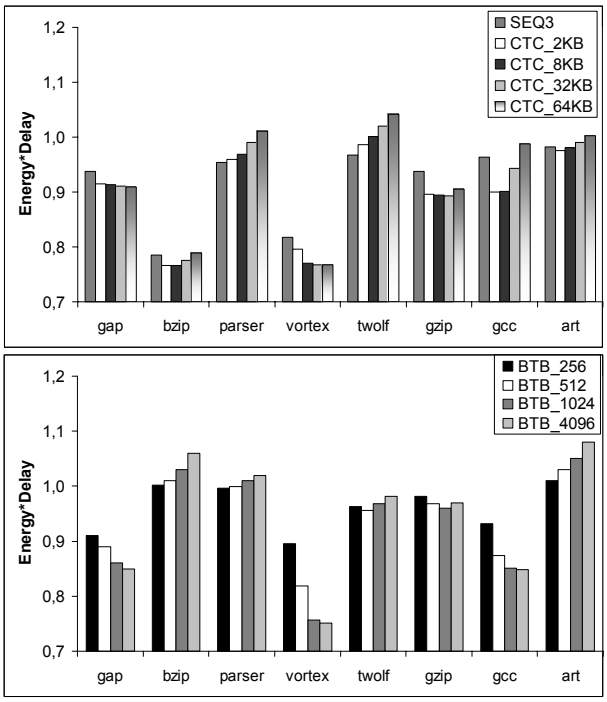

Figure 1. Energy-delay product of program execution under different fetch mechanisms and structure configurations. Figure 1-(a) (top) shows the energy-delay product of SEQ3 and CTC with different trace cache sizes. The results are normalized to SEQ1. Figure 1-(b) (bottom) shows the energy-delay product using different BTB sizes. The results are normalized to a BTB with 128 entries.

From these observations, it is obvious that fetch requirements vary significantly from application to application. So, we propose an adaptive fetch mechanism choosing among different fetch schemes: SEQ1, SEQ3, CTC, and a modified STC (the modification is explained in Section 3.4).

The first two configurations avoid trace cache access while the others make use of it. In some cases when the branch misprediction rate is high, using a trace cache is counterproductive, since it executes many wrong path instructions, which increases pollution in some structures like the cache and increase energy consumption. In these cases it may make sense to use a SEQ1 or a SEQ3 fetch policy. For the trace cache based schemes, our adaptive mechanism has two

possibilities. In the first, CTC, the fetch unit accesses at the same time to the trace cache, the I-cache, the multi-branch predictor, and the BTB (in a similar way to CTC [2, 3]). This configuration is useful when the trace cache behaves well, but the I-cache still provides a considerable amount of instructions. The second configuration is a modified STC. In those cases when trace cache provides almost all instructions to the back-end, the fetch unit virtually divides into two phases, like in [2] and [3]. It accesses first the trace cache and the multi-branch predictor, and then the I-cache and the BTB – only when a trace cache miss occurs.

The adaptive fetch mechanism can also adjust the size of some structures. Figure 1 suggests that the optimal size for trace cache or BTB varies. Consequently, our adaptive fetch mechanism resizes these structures to the most appropriate configuration.

3. ADAPTIVE FETCH MECHANISM

We intend to adapt the fetch policy and trace cache and BTB sizes at the boundary of program phases, where the behavior is about to change. While the concept of program phases is simple at an intuitive level – that the program goes through different periods of execution with different behavior, accurate and efficient phase detection and behavior prediction for future phases are challenging and are currently being investigated by many research groups. In this paper, we use a code-based approach [8] that is straightforward and very effective in our design. We associate the behavior of a period of execution with the static code section that is being executed and use the behavior of past instances of that section to predict the behavior of future instances. While this learning and prediction process can be performed online, the need for any extra hardware for prediction will not only complicate the hardware but also consume energy, which will reduce the benefit of adaptation. In this paper, we adopt an offline approach for its simplicity and effectiveness.

We first need to establish the phase boundaries (Section 3.1). Once each phase is identified, we use profile information to decide what policy and structure sizes to use in each phase. This information is then encoded into the binary, so that at runtime, it instructs the hardware to adapt. Sections 3.2 and 3.3 describe these steps.

3.1 Phase Detection

As mentioned above, we use a simple code-based approach. We partition the code into modules purely based on granularity. We want the modules to be big enough to reduce overhead associated with runtime adaptation and yet small enough to have more consistent behavior. The technique employed is the same to the one detailed in [6, 7]. Arguably, the partitioning may be improved when the behavior of the modules is taken into account. We leave this as future work.

We tie fetch reconfiguration to the static code because, intuitively, the code strongly affects fetch demand. After all, every structure from the fetch mechanism uses an instruction address, exclusively or inclusively, to index it. Also, the runtime path of instructions does not change much. Finally, prior research has shown that tying adaptively control to the code's position is generally more effective than time-based prediction and control mechanisms [8].

The module's granularity is important. If a module is too fine-grain, the reconfiguration overhead at runtime will be large. If it is too coarse-grain, it may contain smaller units with different fetch demands. So it becomes important to find the optimal module-division of the application. In this work, we use important subroutines as modules. Besides, when the subroutine is too big, it is further divided into its internal loops. This process of partitioning follows [7]. The particular thresholds we use in our study in selecting subroutines and loops are the average length per invocation and total execution time weight. We set these thresholds to 1 microsecond and 5 percent respectively. As an example, *gzip* was divided in 7 modules. We found 5 important subroutines, one of them big enough to be further divided, so it was split in 3 internal loops. The resulting partition works well in our study: the adaptation not only results in significant energy savings, but also improves the performance.

The overhead resulted from runtime adaptation is negligible. In our experiments, even assuming a very conservative 50-cycle switching overhead, the percentage of cycles spent in reconfiguration is less than 0.01% in all cases due to the relatively coarse granularity of the modules.

3.2 Configuration Space Exploration

Our approach draws on profiling with reduced inputs to determine each module's demand. The energy and performance metrics are collected by means of software instrumentation, simulation or by using hardware performance counters. A naive implementation would exhaustively search the space, covering all possible combinations of the different module's configurations. However this is impractical because it would require n^m profiling runs, where n is the number of possible processor configurations and m is the number of modules.

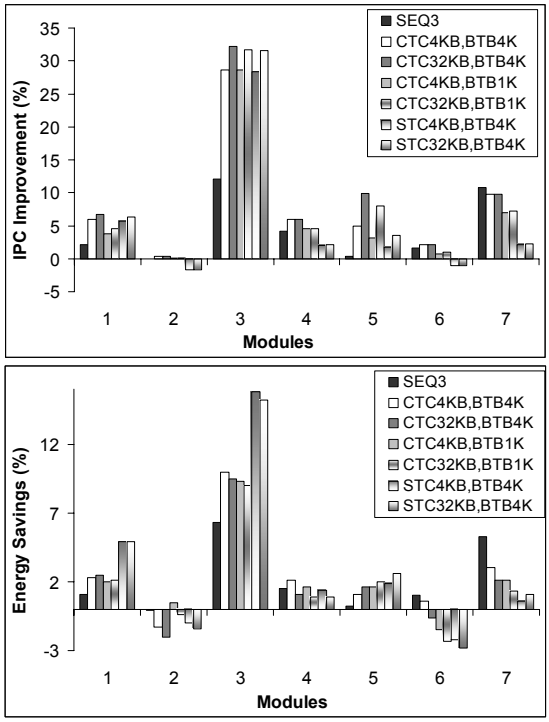

Figure 2. Results for the different modules of gzip. Figure 2-(a) (on top) illustrates IPC improvement of different schemes over a SEQ1. Figure 2-(b) (on bottom) shows energy savings over a SEQ1. A negative value implies performance or energy loss in the given scheme compared to SEQ1.

We decrease the number of experiments assuming that choosing a different configuration for one module does not affect other modules. Under this assumption, the number of experiments in the profiling stage decreases to *n*m*, significantly fewer than the naive solution. The assumption ignores the effect of destructive or constructive interference among different modules. This interference tends to be secondary unless the size of the trace cache becomes very small. Given that we never go down to such small sizes this effect is irrelevant in our study.

As an example, Figure 2 illustrates results for a given application (gzip). Figure 2-(a) illustrates the IPC improvement achieved in each module using a SEQ3 and different CTC and STC schemes over a SEQ1 baseline. Figure 2-(b) shows the processor-wide energy savings achieved for the same set of configurations, compared again to the baseline SEQ1. We can see that the energy savings can be quite dramatic in certain modules, reaching more than 15% *processor-wide* energy savings just by adapting the fetch mechanism.

3.3 Decision Making for Adaptation

After per-module exploration, we have to obtain the best configuration for each module, depending on the energy and performance target. This is the same as solving a knapsack problem [9]: the tolerable performance degradation is the knapsack's capacity while the total energy savings is the value we want to maximize. As in [6], we employ a greedy strategy to find an approximation solution.

The knapsack algorithm adopts a global approach in deciding the configurations to choose for each module. We observe that a simpler approach may also work. For some modules, configurations are very easy to select just based on local information of that module and would not require the more global consideration. For example, from Figure 2, for module 3 it is obvious that an STC is the best approach to use, since it obtains a very similar IPC as the others, but achieves higher energy savings. On the other hand, module 7 should adapt to a SEQ3 configuration, since any of the trace cache based configurations behaves significantly poorer, due to a high number of wrong path speculated instructions.

3.4 Hardware Support

As mentioned above, we intend to adapt the fetch unit to match the requirement of each application phase. Our fetch unit can adopt any of the 4 different configurations that we consider (SEQ1, SEQ3, CTC, and a modified STC). We modify the STC originally proposed by Hu *et al.* [2, 3]. In our design, the BTB is not accessed when the trace cache hits. This can be done since our trace cache directly provides the target address.

In addition to adapting to the best policy, trace cache and BTB sizes can also be adapted. There are several existing schemes for resizing caches and circuitry to perform the resizing. In particular, the associativity [4], the number of sets [5], or the combination of the two can be adjusted dynamically. The effectiveness of these schemes depends on the particular cache structure organization. In our case, dynamically varying the number of sets of the baseline cache is more effective than changing cache ways.

Figure 3 illustrates the design that we are using. When the SEQ1 signal is set, trace cache access is disabled, and just the first prediction bit from the multi-branch predictor is used. When the SEQ3 signal is set, trace cache access is disabled, but the 3 prediction bits are used. When STC_1 is set (first phase of a modified STC configuration), access to I-Cache and BTB is gated, while if STC_2 is set (second phase of a modified STC configuration), access to trace cache and multi-branch predictor is gated.

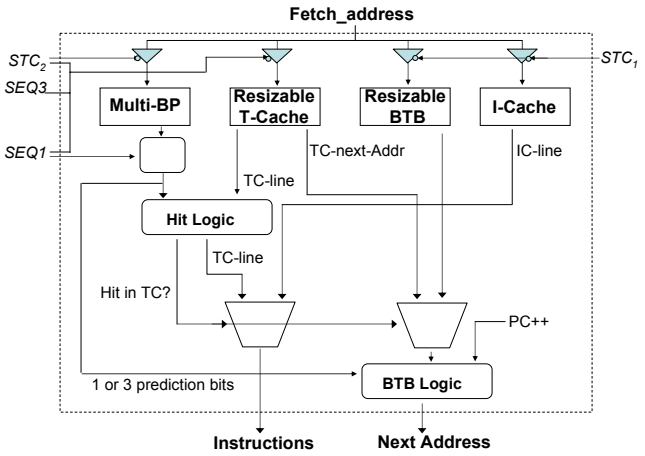

Figure 3. Hardware support.

4. EXPERIMENTAL FRAMEWORK

We have evaluated our proposed adaptive fetch on a simulated generic out-of-order processor, whose main parameters are summarized in Table 1. As the evaluation tool, we employ a modified version of SimpleScalar [10], incorporating the Wattch framework [11] to evaluate energy consumption.

To evaluate our designs for different applications, we select several benchmarks from the SPEC CPU2000 suite. In selecting those applications, we tried to cover the wider variety of behaviour. We simulated each application to completion, using the *train* input as our default production input.

Table 1. Simulated parameters.

16-issue out-of-order processor
120 integer and floating-point physical registers
I-Queue and FP-Queue: 64 entries
Branch predictor: 2-level, 16K entries / BTB: 4K entries
RAS: 32 entries / LSQ: 128 entries
L1 data cache: 64KB, 4-way, LRU, latency: 1 cycle
L2 cache: 512KB, 4-way, LRU, latency: 6 cycle
L1 instruction cache: 64KB, 2-way, LRU, latency: 1 cycle
Trace cache: 32KB, 2-way, LRU, latency: 1 cycle, partial matching
Memory access: 100 cycles

The trace cache we are simulating is accessed by the least significant bits of the fetch address. Each trace contains: a tag (most significant bits of the address), 3 prediction bits, the number of branches and of instructions that the trace contains, a maximum of 16 instructions, and the next instruction PC for a final taken branch. For filling in the trace cache, there is a buffer that stores every committed instruction, so that when a trace is completed, it is stored into the trace cache (at a maximum rate of one trace per cycle). The trace cache we employ allows partial matching [1].

We should highlight that this implementation guarantees one cycle access to the trace cache, but only allows one trace per fetch address.

Our baseline fetch unit uses a 32KB trace cache, a 64KB I-cache, a 4K-entry BTB, and a 2-level multi-branch predictor with 16K entries. Both BTB and trace cache can be downsized to smaller structures (BTB: 4K, 2K, 1K, 512, 256, or 128 entries; trace cache: 32KB, 16KB, 8KB, 4KB, or 2KB).

The multi-branch predictor that we are using is a two level gshare predictor with a golden BHR [12] updated in the fetch stage.The main BHR and PHT are updated in the decode stage.

5. EXPERIMENTAL RESULTS

5.1 Energy Savings

Table 2 summarizes the energy savings in the processor's fetch unit of PBAFM over 4 different non-adaptive designs (SEQ1, SEQ3, CTC-4KB, and CTC-32KB), all of which employ a 4K-entry BTB and a 64KB I-cache.

Table 2. Fetch unit energy savings achieved by PBAFM over 4 different non-adaptive designs (SEQ1, SEQ3, CTC-4K, and CTC-32K).

	Gap	*Bzip*	*Parser*	*Twolf*	*Crafty*
SEQ1	31%	49%	19%	22%	16%
SEQ3	26%	33%	19%	20%	15%
CTC-4KB	24%	36%	26%	26%	20%
CTC-32KB	26%	38%	35%	39%	31%
	Vortex	*Gcc*	*Gzip*	*Vpr*	**INT-Avg**
SEQ1	32%	25%	23%	17%	**26.0%**
SEQ3	27%	21%	21%	15%	**21.9%**
CTC-4KB	26%	20%	21%	22%	**24.6%**
CTC-32KB	28%	20%	19%	34%	**30.0%**
	Applu	*Art*	*Mgrid*	*Swim*	**FP-Avg**
SEQ1	33%	27%	22%	32%	**28.5%**
SEQ3	36%	29%	24%	34%	**30.8%**
CTC-4KB	40%	33%	30%	40%	**35.8%**
CTC-32KB	46%	41%	40%	46%	**43.3%**

As can be noticed, the energy savings achieved by PBAFM are significant both in integer and floating-point applications. All the adaptation mechanisms incorporated in PBAFM contribute to this enhancement. Trace cache and BTB resizing are useful in most cases, especially in floating-point applications. In those modules where trace cache performs poorly, it is disabled, which results in significant energy savings. On the other hand, in those modules where the trace cache performs exceptionally well, the savings come from the use of a modified STC, in which BTB and I-cache are accessed only when the trace cache misses.

As Table 3 illustrates, improvements in the fetch unit translate into notable energy savings in the whole processor. These savings are not only due to the savings in the fetch unit. In those modules where a less aggressive fetch policy is used (especially SEQ1) an extra saving comes from the mis-speculation reduction (mis-speculation pollutes the branch predictor and incurs extra energy executing wrong instructions). In addition, for integer applications, improving performance (as will be shown in Section 5.2) saves energy, mostly by cutting down extra clock distribution energy.

Table 3. Processor energy savings achieved by PBAFM over 4 different non-adaptive designs (SEQ1, SEQ3, CTC-4K, and CTC-32K).

	Gap	*Bzip*	*Parser*	*Twolf*	*Crafty*
SEQ1	5.8%	10.0%	3.1%	3.4%	5.8%
SEQ3	4.5%	5.4%	1.7%	2.0%	2.1%
CTC-4KB	3.1%	5.1%	3.8%	4.1%	4.3%
CTC-32KB	2.5%	6.1%	4.4%	4.8%	4.7%
	Vortex	*Gcc*	*Gzip*	*Vpr*	**INT-Avg**
SEQ1	10.8%	6.9%	7.4%	3.9%	**6.4%**
SEQ3	4.6%	4.0%	4.9%	1.6%	**3.4%**
CTC-4KB	3.9%	2.9%	3.3%	2.2%	**3.6%**
CTC-32KB	3.9%	3.2%	3.5%	2.4%	**3.9%**
	Applu	*Art*	*Mgrid*	*Swim*	**FP-Avg**
SEQ1	3.3%	3.9%	3.2%	3.8%	**3.6%**
SEQ3	2.9%	3.7%	3.0%	4.1%	**3.4%**
CTC-4KB	2.7%	2.7%	3.0%	4.3%	**3.2%**
CTC-32KB	2.6%	2.8%	2.8%	4.6%	**3.2%**

5.2 Performance

Table 4 shows performance improvement achieved when using a PBAFM compared to the 4 non-adaptive designs. A negative value means that performance gets worse in our adaptive approach.

Table 4. Performance improvement achieved by PBAFM over 4 different non-adaptive designs (SEQ1, SEQ3, CTC-4K, and CTC-32K).

	Gap	*Bzip*	*Parser*	*Twolf*	*Crafty*
SEQ1	7.6%	16.1%	3.8%	2.3%	6.5%
SEQ3	3.5%	2.2%	0.3%	0.0%	0.7%
CTC-4KB	1.7%	0.7%	0.6%	1.1%	2.9%
CTC-32KB	1.1%	0.7%	0.6%	1.0%	2.9%
	Vortex	*Gcc*	*Gzip*	*Vpr*	**INT-Avg**
SEQ1	17.3%	7.0%	8.8%	3.9%	**8.1%**
SEQ3	5.1%	4.3%	3.7%	0.2%	**2.2%**
CTC-4KB	1.5%	0.5%	0.9%	0.3%	**1.1%**
CTC-32KB	0.4%	0.5%	0.7%	0.3%	**0.9%**
	Applu	*Art*	*Mgrid*	*Swim*	**FP-Avg**
SEQ1	0.3%	0.6%	-0.1%	-0.3%	**0.1%**
SEQ3	0.3%	0.3%	-0.1%	-0.3%	**0.1%**
CTC-4KB	0.0%	0.1%	0.0%	-0.2%	**0.0%**
CTC-32KB	0.0%	0.1%	0.0%	-0.3%	**0.0%**

Using PBAFM, we obtain performance improvements in most integer applications. Each module employs an optimal fetch configuration, which provides sufficient supply of instructions and yet without using an overly aggressive policy that can introduce many wrong-path instructions and negatively impacting performance. Moreover, in modules where a small trace cache or BTB is sufficient, keeping the structure size small also helps to reduce conflict in the disabled portion of the structures, which indirectly benefits the execution of other modules. For some modules of the floating point applications, the trace cache is simply not effective, so PBAFM ends up disabling it in these cases. As can be seen from Table 4, the performance implication is thus negligible.

6. RELATED WORK

Researchers have proposed other solutions to improve the fetch unit management. Most of these proposals include a trace cache

structure in the fetch stage, and our proposal can work on top of these designs to further improve the fetch mechanism.

In [2] and [3], Hu *et al.* propose two new models for a fetch stage. The first model, which they call *Selective Trace Cache*, uses compiler and hardware support to control trace cache lookup (avoided in the cases where trace cache behaves poorly). This approach can achieve some energy reduction in the fetch stage, but at the cost of some performance loss, compared to a CTC. A second approach proposed in this prior work is a *Direction Predictor based Trace Cache* (DPTC). In this case, the selection for the fetch unit configuration is dynamic, eliminating the need for recompilation and ISA modification. However, this comes at the cost of some overhead. The model can reduce energy consumption in the fetch mechanism, but again with some performance loss compared to CTC.

In [13], Buyuktosunoglu *et al.* introduce a scheme based on a combination of fetch gating and issue queue adaptation to jointly adapt the fetch and issue stages so as to match the current parallelism characteristics of the application.

In [14], Santana *et al.* propose software techniques to reorganize the code so that the fetch engine complexity can be reduced. In a similar way, Ramirez *et al.* [15] also propose compiler optimizations to improve the layout of instructions.

In [16], Co and Skadron perform a set of fetch engine area and associativity experiments as well as a next trace predictor design space exploration.

In the broader domain of low-power design, some researchers have proposed structure resizing for achieving energy savings. In particular, the associativity [4] or the number of sets [5] of a cache can be adjusted dynamically to the most convenient configuration. We have applied some of these techniques to resize trace cache and BTB when needed.

Finally, in our prior work, we demonstrated the effectiveness of an adaptive design of the branch prediction, including the direction predictor and the BTB [6].

7. CONCLUSIONS

In this paper we have proposed PBAFM, a new adaptive fetch mechanism that adjusts the underlying hardware to meet the application's demands. Our design switches to less expensive configurations when appropriate, resulting in significant energy savings in the fetch unit. In doing so, the performance is not compromised. In fact, for integer applications, we even obtain modest performance gain relative to a very aggressive fetch unit. Comparing PBAFM with the most aggressive fetch unit design studied (a 32KB CTC), energy consumption in the fetch unit is reduced by 34.1% on average and the performance is improved by 0.6%. The combined effect is a 3.7% processor-wide energy reduction. Compared to a baseline fetch unit (SEQ1), PBAFM improves performance by 5.7%, while saving 26.8% and 5.5% energy in the fetch unit and the whole processor, respectively.

8. REFERENCES

[1] E. Rotenberg, S. Bennett, and J. E. Smith. Trace Cache: A low latency approach to high bandwidth instruction fetching. *International Symposium on Microarchitecture*. November, 1996.

[2] J. S. Hu, N. Vijaykrishnan, M. J. Irwin, M. Kandemir. Optimizing Power Efficiency in Trace Cache Fetch Unit. *Technical Report*, Department of Computer Science and Engineering, Pennsylvania State University, 2003.

[3] J. S. Hu, N. Vijaykrishnan, M. J. Irwin, M. Kandemir. Using Dynamic Branch Behavior for Power-Efficient Instruction Fetch. *International Symposium on VLSI*, February, 2003.

[4] D. H. Albonesi. Selective Cache Ways: On-Demand Cache Resource Allocation. *Journal of Instruction-Level Parallelism*, Vol. 2. May, 2000.

[5] S. Yang, M. D. Powell, Babak Falsafi, K. Roy, and T. N. Vijaykumar. An Integrated Circuit/Architecture Approach to Reducing Leakage in Deep-Submicron High-Performance I-Caches. *International Symposium on High-Performance Computer Architecture*, January, 2001.

[6] M. C. Huang, D. Chaver, L. Pinuel, M. Prieto, and F. Tirado. Customizing the Branch Predictor to Reduce Complexity and Energy Consumption. *IEEE Micro* 23(5):12-25, September 2003.

[7] Wei Liu, and M. C. Huang. EXPERT: Expedited Simulation Exploiting Program Behavior Repetition. *International Conference on Supercomputing*, June 2004.

[8] M. C. Huang, J. Renau, J. Torrellas. Positional Adaptation of Processors: Application to Energy Reduction. *International Symposium on Computer Architecture*, June 2003.

[9] T. Cormen, C. Leiserson, and R. Rivest. Introduction to Algorithms. *McGraw-Hill*, 1989, pp. 333-336.

[10] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *Computer*, 35(2), February 2002.

[11] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations., *International Symposium on Computer Architecture*, July, 2001.

[12] T. Y. Yeh and Y. N. Patt. Alternative Implementations of Two-Level Adaptive Branch Prediction. *International Symposium on Computer Architecture*, May, 1992.

[13] A. Buyuktosunoglu, T. Karkhanis, D. H. Albonesi, and P. Bose. Energy Efficient Co-Adaptive Instruction Fetch and Issue. *Computer Architecture News*, Vol. 31. May, 2003.

[14] O. J. Santana, A. Ramirez, M. Valero. Reducing Fetch Architecture Complexity Using Procedure Inlining. *INTERACT-8*, Madrid, Spain. February 2004.

[15] A. Ramirez, O. J. Santana, J. L. Larriba-Pey, M. Valero. Fetching instruction streams. *International Symposium on Microarchitecture*, November, 2002.

[16] Michele Co and Kevin Skadron. Evaluating the Energy Efficiency of Trace Caches. *Technical Report* CS-2003-19, University of Virginia, 2003.