Appears in IEEE Micro 23(5):12-25, Sep./Oct. 2003

Customizing the Branch Predictor to Reduce Complexity and Energy Consumption

TO EXPLOIT INSTRUCTION-LEVEL PARALLELISM, HIGH-END PROCESSORS USE BRANCH PREDICTORS CONSISTING OF MANY LARGE, OFTEN UNDERUTILIZED STRUCTURES THAT CAUSE UNNECESSARY ENERGY WASTE AND HIGH POWER CONSUMPTION. BY ADAPTING THE BRANCH TARGET BUFFER'S SIZE AND DYNAMICALLY DISABLING A HYBRID PREDICTOR'S COMPONENTS, THE AUTHORS CREATE A CUSTOMIZED BRANCH PREDICTOR THAT SAVES A SIGNIFICANT AMOUNT OF ENERGY WITH LITTLE PERFORMANCE DEGRADATION.

Michael C. Huang
University of Rochester

Daniel Chaver
Luis Piñuel
Manuel Prieto
Francisco Tirado
Universidad Complutense de Madrid

• • • • • • Computer systems' prevalence in almost every aspect of society has profound implications for computer system designers. With the number of computers in use (the industry is shipping about a quarter billion CPUs a year just for PCs and servers), their energy efficiency has a significant impact on the economy and the environment. On a smaller scale, high energy consumption in modern microprocessors complicates many system design issues, such as battery life, heat dissipation, and electricity delivery.

High-end processors typically incorporate powerful branch predictors consisting of many large structures that together consume a large portion of total chip power. Depending on the application, some of these structures are underutilized for long periods or contribute nothing to the prediction. This results in energy waste and high power consumption. Here, we propose to gauge branch prediction demand and dynamically adjust prediction resources accordingly. Specifically, we customize the size of the branch target buffer (BTB) and the composition of the hybrid direction predictor for each code section. To gauge branch prediction demand, we use a profile-based approach that incurs little runtime overhead and results in simple and straightforward architectural support. The approach is also very accurate because of program behavior repetition.

We focus on reducing branch prediction energy consumption in high-performance processors by dynamically reducing the branch predictor's complexity. To clarify our concept of *complexity*, we informally classify it into two

Published by the IEEE Computer Society

categories: *Design complexity* describes the design's static, conceptual complexity—that is, how difficult it is to understand, verify, and test the design. *Operational complexity* describes a more dynamic aspect of the design—the extent of the circuit (the number of components or the magnitude of capacitive load) that is actively switching to perform a particular operation. Operational complexity relates directly to energy consumption.

Although the two concepts are correlated, they do not necessarily change in the same direction. In fact, there is a somewhat intrinsic tradeoff between the two types of complexity, especially for high-performance architectures. To maintain certain performance levels, queues or buffers must be very large. As their size increases, these structures are more often designed as clusters or hierarchies of smaller components rather than large, monolithic structures.[1] This type of design reduces operational complexity at the expense of some extra design complexity.

Gating is another typical method of reducing operational complexity, albeit dynamically. With gating, all the active circuit components contribute to energy consumption, but not all of them actually contribute to the operation. For example, in a typical set-associative cache, multiple data and tag ways are accessed in parallel. Although one cache way at most contains the data of interest, energy is spent in all cache ways. Explicitly separating (or gating) noncontributing segments from the rest of the circuit reduces operational complexity in terms of capacitive load or switching activity. However, the circuitry that handles the gating adds to design complexity and thus should be very simple. An example of this separation is clock gating, a technique used extensively in recent systems. This technique consists of gating the clock signal at a certain level of noncontributing functional blocks to prevent the circuit downstream from switching and consuming energy. However, clock gating sometimes requires extra clock buffer and control signals, which can potentially negate the energy savings if the gating is too fine grained.

We have developed a technique in which we apply structure resizing and access gating to create a customized branch predictor. Branch prediction is a key technology in exploiting instruction-level parallelism. Modern high-end processors use an array of tables for branch direction and destination prediction.[2] These tables are large (more than 350,000 bits just for the direction predictors in the Alpha EV8) and are accessed every cycle. This high operational complexity causes significant energy consumption: In certain cases, branch prediction accounts for more than 10 percent of total chip power. High prediction accuracy is essential to high performance and energy efficiency. Mispredictions cause the processor to waste energy executing wrong path instructions. They also increase execution time and related energy overhead such as energy for clock distribution.

However, always using the branch predictor's maximum configuration, regardless of application demands, is not energy efficient. Consider a loop-based application with a very regular control flow: A simple branch predictor—or even simply predicting all branches taken—would work just as well as a complex hybrid predictor. Our idea is to bring the general principle of on-demand resource allocation to branch prediction by using software to customize the predictor according to its resource demands. To achieve this goal with little added energy overhead or design complexity, we propose a design that moves the responsibility of identifying and expressing resource demands to software components.

This on-demand branch prediction, which we call *adaptive branch prediction*, incurs two types of overhead: reconfiguration overhead and energy waste incurred by increased misspeculation due to weaker predictor configurations. To reduce such overhead, we adopt a feedback-based approach. We divide an application into smaller units called *modules*, characterize their branch prediction demand through profiling, and then instrument the application to dynamically reconfigure the predictor. The benefit is twofold: First, because characterization and decision making take place offline, this approach incurs little runtime reconfiguration overhead and adds little to the processor's design complexity. Second, because of the modules' behavior repetition, the hardware receives accurate demand information, which leads to a highly efficient reduction of operational complexity.

Branch predictor reconfiguration

To reconfigure the branch predictor at the

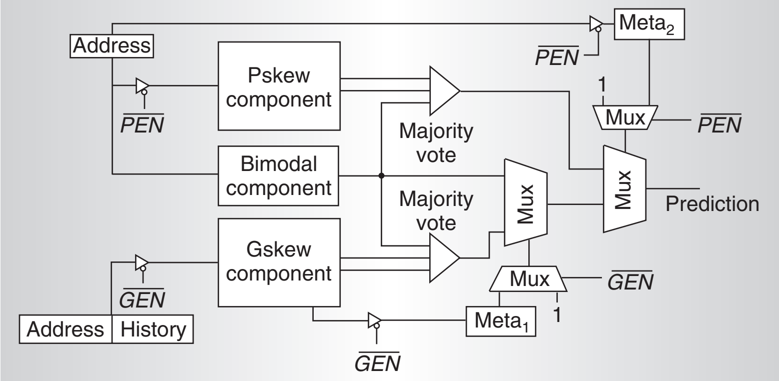

Figure 1. Adaptive hybrid predictor with gating controls. For simplicity, we do not show some structures, such as the local history table for the pskew component.

circuit level, we use access gating to reduce ineffectual circuit switching and table resizing to reduce the capacitive load. There are other ways to reconfigure the branch predictor. We focus on these two generic, straightforward methods, mainly to demonstrate the effectiveness of our overall approach. Modern branch predictors contain numerous table structures, and there are many ways to apply the access-gating and resizing techniques to different tables. To limit our exploration, we use a specific circuit-level technique from each category. For access gating, we use a simple extension to the direction predictor. For table resizing, we focus on the BTB and apply Yang et al.'s technique,[3] one of many for reconfiguring cache-like structures.[4]

Adaptive hybrid predictor

Our baseline direction predictor is a dealiased hybrid predictor named 2Bc-gskew-pskew.[5] Like many other hybrid predictors, this one accesses various prediction tables, and the resulting predictions go through majority voting—sometimes a series of majority votes. For certain branches, a vote from a specific prediction table or tables can be more accurate than a majority vote. Metatables are designed to decide which vote prevails.

To reduce energy waste in accessing multiple tables, we disable tables that do not contribute much prediction accuracy improvement. As Figure 1 shows, we decompose the baseline predictor into three components: gskew, pskew, and bimodal. We use control signals $\overline{GEN}$ and $\overline{PEN}$ to disable gate accesses to the gskew and pskew components. We can wire the signals to a special control register and set or clear them with special register load instructions. Once this mechanism disables a subset of tables, the tables remain inactive until enabled again. When a certain component is disabled, related metatables may become irrelevant and can be disabled as well. For example, when $\overline{PEN}$ is asserted, the pskew component and metatable Meta$_2$ are gated, and the rest of the tables essentially form a gskew predictor.

We do not gate the bimodal component because it also supports majority voting in the other two components and does not consume a significant amount of energy. Moreover, having only two control signals simplifies the circuitry.

In theory, it is possible to gate accesses to branch prediction tables for every prediction by using the metatables to allow accesses only to tables whose predictions will be selected. This saves the energy wasted in accessing tables whose predictions will be disregarded anyway. In practice, this does not work well because it sequences accesses to the metatable and the prediction tables, significantly slowing branch prediction.

Adaptive BTB

Accurate branch *direction* prediction is not enough; without the target address, instruction fetch cannot proceed after predicted-taken branches. The BTB helps provide the target address quickly. Increasing the BTB's size and associativity helps reduce conflict and capacity misses. However, for certain applications, large size can be wasteful. For example, the compress benchmark in the SPEC95 suite has only 46 static branch instructions.[2] Therefore, we resize the BTB on the fly. The BTB is similar to a normal data cache in terms of organization and access. There are several schemes for resizing caches and circuitry to perform the resizing. In particular, the associativity,[4] the number of sets,[3] or the two combined can be adjusted dynamically. These schemes' effectiveness depends on the particular cache structure organization. In our case, dynamically varying the number of sets of our baseline BTB is more effective than changing BTB ways.[6] We use instructions to select the desired size (a power of two) through special control registers, which trigger hardware to adjust the BTB accordingly.

In the context of data caches, resizing necessitates flushing blocks or other mechanisms to maintain coherence. In our case, flushing is not necessary because the information stored in any prediction table affects only performance and energy, not program execution correctness.

Control policy: profiling approach

Adaptive architecture is a promising technology for meeting applications' diverse and dynamic resource demands efficiently. Nevertheless, managing the adaptation of branch predictors is a challenging task. We must balance costs and benefits carefully. We can switch to a less power-hungry branch predictor configuration only if the switch causes minimum degradation of prediction accuracy. Processing wrong path instructions causes potentially far more energy waste in other parts of the processor than is saved in the branch predictor. Another problem is that any extra hardware for keeping track of and predicting branch prediction demand will itself consume extra energy, directly cutting savings. Consequently, we use a profile-based feedback mechanism to estimate branch prediction demand without incurring runtime overhead.

Module selection. We statically partition the code into smaller sections, called modules. A module is the smallest unit to which we apply branch prediction reconfiguration. We tie branch predictor reconfiguration to the static code because, intuitively, the code strongly affects branch prediction demand. After all, the predictor uses an instruction address, exclusively or inclusively, to index almost every table. Also, the runtime characteristics of branches (and thus the appropriate branch predictors) do not change much. Certain branches are strongly biased; others correlate with other branches. Finally, our prior research has shown that tying behavior prediction, and thus adaptivity control, to the code's position is generally more effective than time-based prediction and control mechanisms.[7]

A module's granularity is also important. If a module is too fine grained, the reconfiguration overhead at runtime will be large, reducing energy savings. If it is too coarse grained, it can contain smaller units with different demands on the branch predictor. Here, we select important subroutines as modules. Our experience shows that typical programs are well structured, and performing adaptations at subroutine boundaries is a good engineering solution. To reduce reconfiguration overhead, we use two thresholds in selecting subroutines: average length per invocation, Th_{grain}, and total execution time weight, Th_{weight}. We set these thresholds to 1 microsecond and 5 percent, respectively. We treat the subroutines not selected as extended portions of their dynamic callers.

Per-module exploration. After identifying the module boundaries, we perform profiling to determine each module's demand for prediction resources, using a training input. We use a straightforward approach, running the application multiple times to directly measure the performance and energy effect of different predictor configurations for each module. During this stage, we can obtain the energy and performance metrics by various methods: software instrumentation, simulation, or using hardware counters. In a naive implementation, we would exhaustively search the space, covering all combinations of configurations for each module. But this is impractical because it would require n^m profiling runs, where n is the number of possible branch prediction configurations and m is the number of modules.

Several approximations can greatly reduce the number of experiments. First and foremost, we assume that choosing a different prediction configuration for one module does not affect other modules, and that the impact of reconfiguring different modules is additive. Under this assumption, the number of experiments in the profiling stage decreases to around $n \times m$, 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 becomes very small. As we explain later, moderate resizing and access gating provide the most benefits, and thus we will not likely select very small sizes. Similarly, we treat the adaptive BTB and hybrid predictor as two independent reconfigurations whose effects are additive.

Second, as we progressively reduce branch prediction resources, performance degradation

Table 1. System configuration.

Processor	Caches	Bus and memory
Six-issue, 1-GHz, out-of-order I-window size: 96 entries Units: 5 integer, 4 floating-point, 2 load/store, 1 branch Pending loads: 16; stores: 16 Branch penalty: 8 cycles Return address stack entries: 32 BTB: 2-way, 4,096 entries Predictor: 2Bc-gskew-pskew	L1 data cache: 32-Kbyte, 2-way, least-recently-used Occupancy: 1; RT: 3 ns L2 cache: 512-Kbyte, 8-way, pseudo-least-recently-used Occupancy: 4; RT: 12 ns Instruction cache: 32-Kbyte, 2-way	Front-side bus: 128-bit, 333-MHz Memory: Two-channel Rambus DRAM bandwidth: 3.2 Gbytes/s Memory RT: 108 ns

RT: contention-free round trip from the processor

increases and energy savings decrease for each module. Thus, we can stop exploration of the next-lower configuration if the current relative improvement in energy savings falls below a certain threshold (for example, 0.1 percent of total processor energy consumption), or if performance degradation goes above a certain threshold (we set it to 0.5 percent of total execution time in this article). This helps prune out unpromising configurations.

Choosing the configurations. After per-module exploration, we can obtain the best configurations by solving the knapsack problem.[8] However, we don't try to solve the problem exactly. Rather, we approximate the best solution using the simple greedy strategy. In this case, the tolerable performance degradation is the knapsack's capacity, and the total energy savings is the value we want to maximize. We have a maximum of $m(n_{\mathrm{BTB}} + n_{\mathrm{HP}})$ items to choose from; m is the number of modules, and n_{BTB} and n_{HP} are the numbers of possible states in the adaptive BTB and the adaptive hybrid predictor. Each item, uniquely identified by a module identification and one of the n_{BTB} possible BTB states or n_{HP} possible direction predictor states, represents the additional energy savings and performance degradation incurred by a module's changing from one state to the next most efficient state.

We use these incremental values in the knapsack problem. To obtain them, for each adaptive technique, we take the energy and performance impact of all the states (for example, n_{BTB}), rank them according to energy savings per unit increase in execution time, and subtract the energy and performance impact of the state ranked immediately above. (To simplify the math, if the performance degradation of a state is negative, which is rare if even possible, we simply use a positive infinitesimal value instead.) If, after the subtraction, the energy savings becomes negative, we remove the state from the ranking. After this process, we will be dealing with a list of pairs of incremental energy savings and performance degradation, both positive. (If, after the subtraction, the energy savings is negative while performance degradation is positive, going from the previous to the current state causes energy waste and slowdown, clearly undesirable. Also, if energy savings and performance degradation are negative after subtraction, the state is undesirable and should be removed. Furthermore, because the list is sorted by decreasing ratio of energy savings to performance degradation, there will not be a state in which additional performance degradation is negative while additional energy savings is positive.)

Evaluation

We evaluated our proposed customized branch predictor on a simulated, generic, out-of-order processor, loosely modeled on a MIPS R10000.[9] The baseline branch predictor was a 2Bc-gskew-pskew predictor configured with

- two 4,096-entry metatables,
- a 4,096-entry bimodal table,
- a gskew component consisting of two 4,096-entry global history tables that use 10 bits of global history, and
- a pskew component consisting of a 1,024-entry local history table (8 bits wide) and two 2,048-entry pattern history tables.

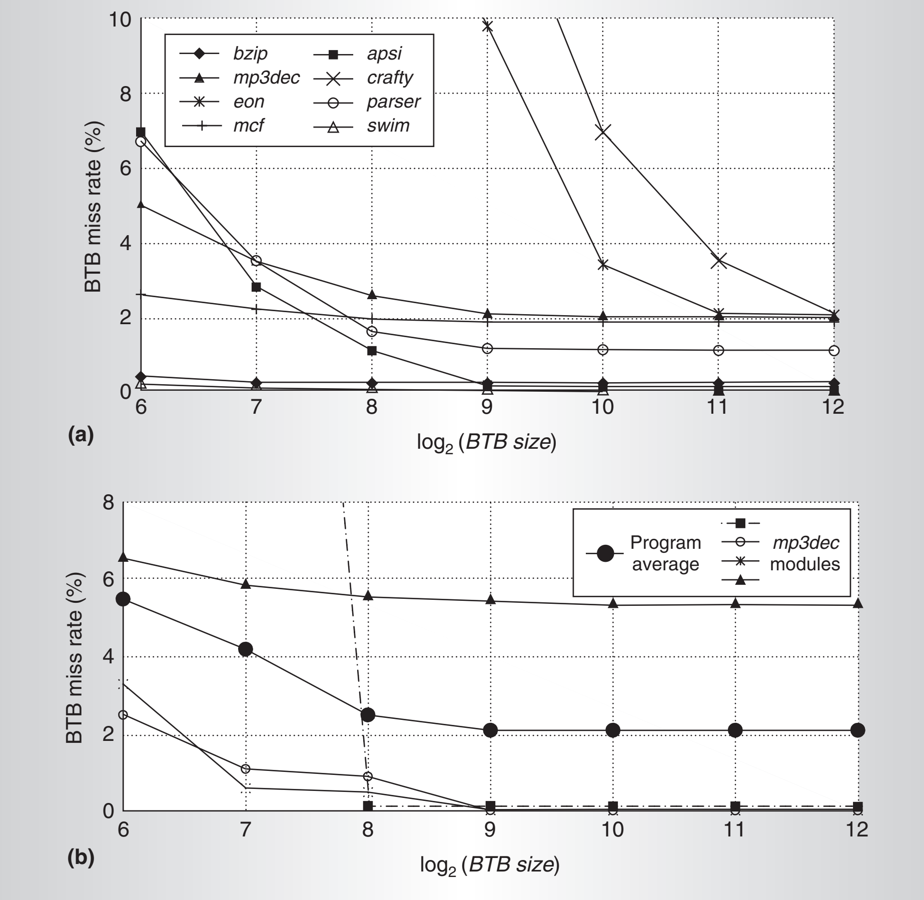

Figure 2. BTB miss rate with different BTB sizes for all applications (a) and for different modules and the program average in the application *mp3dec* (b).

Table 1 shows some of the simulated system's parameters. As the evaluation tool, we use a heavily modified, execution-driven simulator, based on the MIPS Interpreter (MINT), that models all resources' contention and occupancy.[10] The simulator incorporates the Wattch framework[11] to evaluate energy consumption. Our simulator accounts for energy overhead induced by misspeculation.

To evaluate our approach for different types of applications, we selected eight applications, including multimedia (*mp3dec*), integer (*bzip*, *crafty*, *eon*, *mcf*, and *parser*), and floating-point applications (*apsi* and *swim*). The integer and floating-point applications come from the SPEC CPU2000 suite. In selecting applications, we tried to cover a wide variety of behavior. Because it would require an enormous number of simulations, we could not perform our study on a complete benchmark suite.

We compiled the applications with the MIPSPro compiler, using proper optimizations. We simulated each application to completion after skipping the initialization phase, which we identify manually. To make the simulations of the SPEC CPU2000 applications more manageable, we reduced the official SPEC *ref* input set to serve as our default production input.[12] For profiling (training), we used the SPEC *test* input set for these applications. For the multimedia application, we used random music and voice clips as input sets. Simulation length ranged from several hundred million to more than 2.5 billion instructions.

Adaptive BTB

The adaptive BTB exploits the fact that many BTB entries are underutilized. Figure 2 shows the relationship between BTB size and the miss rate for different applications and for

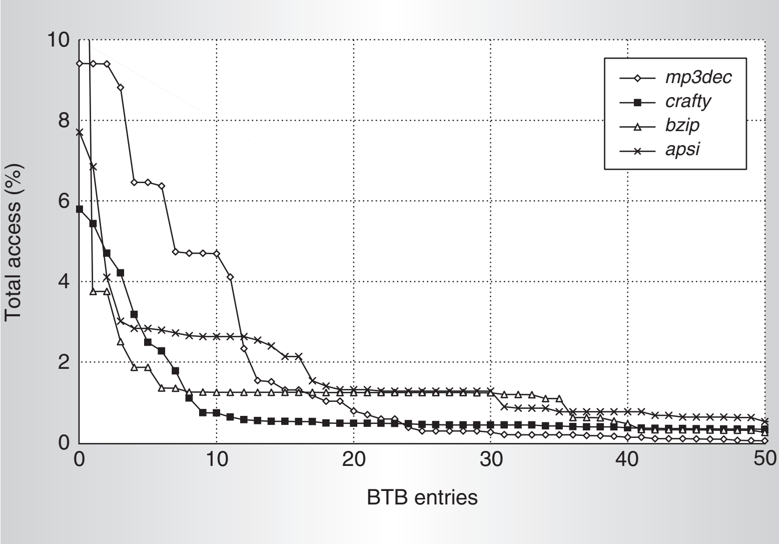

Figure 3. Percentage of total accesses to each entry for a 4,096-entry BTB. The entries are sorted by the number of accesses, and we show only the first 50 entries and four applications. The other applications exhibited very similar results.

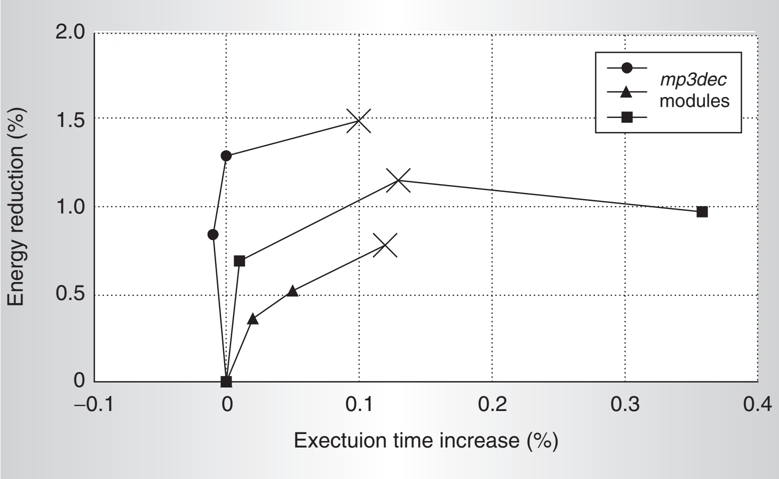

Figure 4. The influence on overall performance and energy consumption caused by progressive resizing of the BTB portions on three modules of *mp3dec*. Each point on the curves corresponds to a different configuration (size). The × marks represent the chosen configurations.

different modules in one application. Figure 2a shows that in some applications, such as *bzip*, the BTB miss rate is almost independent of BTB size, whereas other applications, such as *crafty*, exhibit a significant increase in miss rate for small BTB sizes. This demonstrates the difference in demand at the application level. In addition, as Figure 2b illustrates, BTB demand also varies widely among different modules in a given application. This suggests that in some applications, many BTB entries remain unused for long periods. In Figure 3, we sort BTB entries by the number of accesses to each entry. As the figure shows, a handful of entries are heavily accessed, whereas many others are not.

We followed the profiling procedure outlined earlier to characterize the energy and performance impact of various BTB configurations, using the training input set. Figure 4 illustrates some results of this approach for the application *mp3dec*. The horizontal axis shows the program's relative execution time increase; the vertical axis shows the processor's energy savings relative to the total program's energy consumption. We observe the following: First, for certain modules, moderate BTB resizing can produce relatively significant energy reduction without incurring much slowdown. In Figure 4, each of the three selected configurations results in a total chip energy reduction of about 1 percent, while incurring a slowdown of only about 0.1 percent. Second, beyond a certain size, further reducing BTB size is counterproductive: When the BTB becomes too small, misprediction-induced energy waste outweighs the BTB's extra energy savings. This is demonstrated in Figure 4 by the module whose curve bends downward, indicating that the next configuration incurs more slowdown and reduces energy savings.

To demonstrate the effectiveness of BTB resizing, we set up a 0.5 percent threshold for tolerable performance degradation. We chose such a small threshold to keep systemwide energy overhead low. Using the profile information and this threshold, our offline decision algorithm chose the configuration that saves the most energy without going over the threshold for each application and embedded the decision into the application binaries. We then performed two production runs, using the training input and the production input. Table 2 shows the results. The table does not explicitly show the improvements in various metrics such as *ED* (energy-delay product) or *ED*^{2}. However, because actual performance degradation is generally very small, the improvements in these metrics largely follow the energy consumption reduction.

For each application, the two columns correspond to the training and production input sets used in the production runs. Each col-

Table 2. Energy savings and performance degradation for all applications, using the adaptive BTB.

	apsi		*bzip*		*crafty*		*eon*		*mcf*		*mp3dec*		*parser*		*swim*	
Metric	**T**	**P**	**T**	**P**	**T**	**P**	**T**	**P**	**T**	**P**	**T**	**P**	**T**	**P**	**T**	**P**
$-\Delta E_{\text{Total}}$ (%)	3.55	3.60	8.54	8.62	2.25	2.37	1.60	1.75	7.60	7.72	3.87	3.24	6.54	6.60	1.27	1.17
$-\Delta E_{\text{BP}}$ (%)	57.0	57.8	66.2	66.0	20.1	21.1	43.3	44.8	71.3	72.2	64.6	57.1	57.9	58.4	53.1	49.5
ΔT (%)	0.01	0.01	–0.02	0.08	0.50	0.64	0.75	0.97	0.02	0.02	0.38	0.44	0.05	0.08	–0.01	0.00

T: training input; P: production input

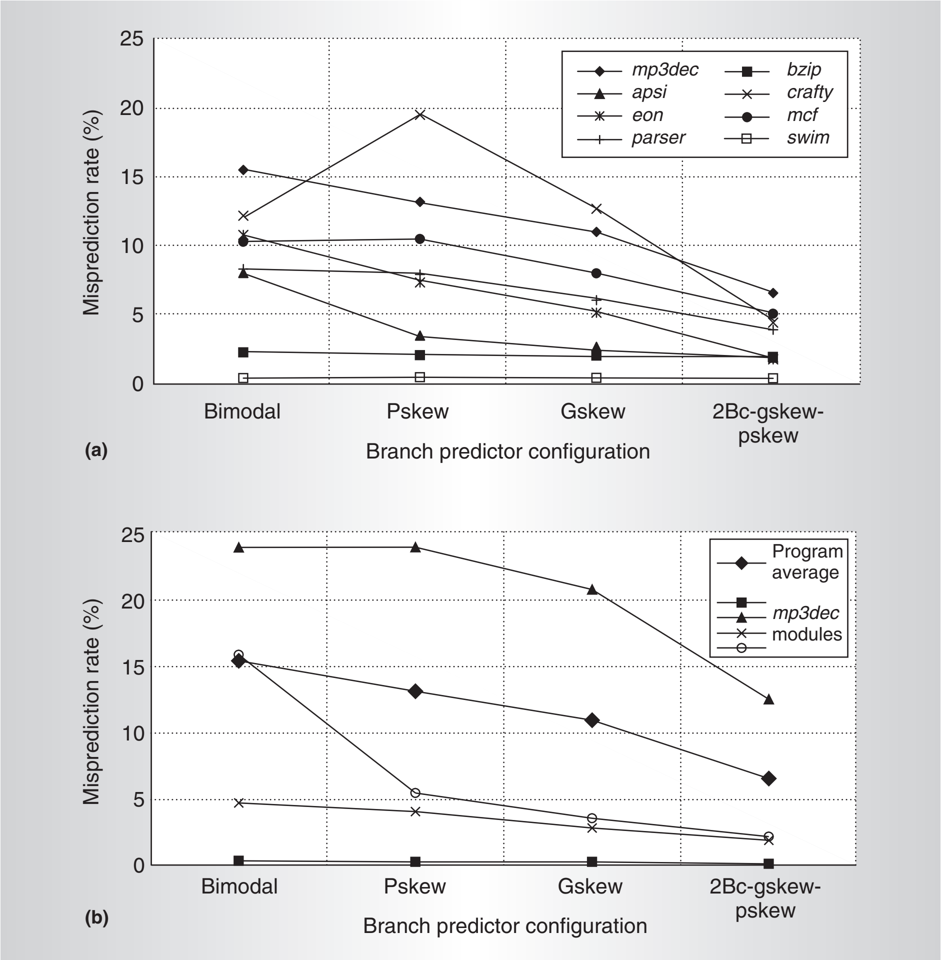

Figure 5. Branch misprediction rate with different branch predictor configurations for all applications (a) and for each module in application *mp3dec* (b).

umn lists the relative energy savings in the processor ($-\Delta E_{\text{Total}}$) and the branch predictor ($-\Delta E_{\text{BP}}$) and the relative increase in total execution time (ΔT). This table shows that dynamically changing BTB size can be profitable: It can save 20 to 70 percent of energy spent in the branch predictor and up to 8.6 percent of total chip energy, with very little performance degradation.

Adaptive hybrid predictor

Just as the demand on the BTB varies both within and across applications, the demand on the direction predictor's strength also varies. Figure 5 shows the misprediction rate for predictor configurations ranging from the most sophisticated full configuration to the simplest bimodal predictor. Figure 5a shows the difference in misprediction rate across

Table 3. Energy savings and performance degradation for all applications, using the adaptive hybrid predictor.

	apsi		*bzip*		*crafty*		*eon*		*mcf*		*mp3dec*		*parser*		*swim*	
Metric	**T**	**P**	**T**	**P**	**T**	**P**	**T**	**P**	**T**	**P**	**T**	**P**	**T**	**P**	**T**	**P**
$-\Delta E_{\text{Total}}$ (%)	1.84	2.11	2.78	2.95	2.43	2.90	0.92	0.75	3.39	3.29	1.51	1.44	2.20	2.10	0.58	0.56
$-\Delta E_{\text{BP}}$ (%)	28.5	32.1	25.3	28.5	21.3	25.0	17.6	17.6	22.7	21.9	24.1	23.5	19.0	18.2	24.2	24.0
ΔT (%)	0.09	0.12	0.03	0.07	0.32	0.29	-0.09	0.35	–0.02	0.01	0.11	0.19	–0.14	–0.01	0.00	0.00

T: training input; P: production input

Table 4. Energy savings and performance degradation for all applications using both the adaptive BTB and the adaptive hybrid predictor.

	apsi		*bzip*		*crafty*		*eon*		*mcf*		*mp3dec*		*parser*		*swim*	
Metric	**T**	**P**	**T**	**P**	**T**	**P**	**T**	**P**	**T**	**P**	**T**	**P**	**T**	**P**	**T**	**P**
$-\Delta E_{\text{Total}}$ (%)	5.20	5.33	11.5	11.5	4.83	5.21	2.26	2.28	10.7	10.8	5.06	4.81	8.15	8.21	1.76	1.81
$-\Delta E_{\text{BP}}$ (%)	78.9	80.5	89.2	89.5	41.6	43.4	54.1	54.4	74.8	75.5	86.1	82.4	72.4	73.2	74.6	75.0
ΔT (%)	0.12	0.15	0.00	0.16	0.82	0.96	0.71	1.26	0.03	0.03	0.55	0.69	–0.02	0.11	0.00	0.00

T: training input; P: production input

applications, and Figure 5b shows the difference across modules in one application.

Figure 5 shows that certain predictors sometimes produce results close to those of the more sophisticated predictors. For example, the bimodal predictor produces satisfactory results for *bzip*, exhibiting a small overall misprediction rate. Experiments show that disabling gate accesses to the gskew and pskew components in the predictor results in notable energy savings (3 percent of total processor energy) without measurable performance degradation for *bzip*. Table 3 summarizes the performance and energy impact of the adaptive predictor for all the applications. Again, we set a 0.5 percent performance degradation threshold.

Combining the two adaptations

Finally, we combined the two techniques to achieve more flexible on-demand branch prediction. Table 4 shows energy savings and performance degradation using both adaptive techniques to save as much energy as possible without slowing down more than 0.5 percent. Again, we based the decision of what adaptation to use on the profiling results using the training input. The table shows that we achieved a reduction in processor-wide energy consumption of about 6.2 percent on average and 11.5 percent maximum.

From the results we've presented, we make a key observation: Using the same profile (based on the training input), we obtained very similar results for the two sets of production runs using different inputs. This confirms our intuition that branch prediction demand largely depends on the code. Also, our results suggest that the two techniques are largely independent under the tested scenario. Finally, we observe that overall energy savings depend not only on the proposed adaptive system's effectiveness but also on the application's branch prediction demand and the proportion of energy spent in branch prediction. For example, the application *crafty* has a very complex control flow and requires powerful branch prediction, so our techniques can achieve only moderate strength reduction for this application. On the other hand, *swim* is a very regular application with many loops, so a very simple predictor configuration can still do a good job of predicting branches. However, floating-point units and the memory subsystem use much of the energy. Thus, even a significant energy reduction in the branch predictor becomes insignificant in the processor as a whole.

Other experimental results

We performed additional experiments to investigate several other issues.

Invocation variation. We propose dynamically adjusting branch prediction strength for

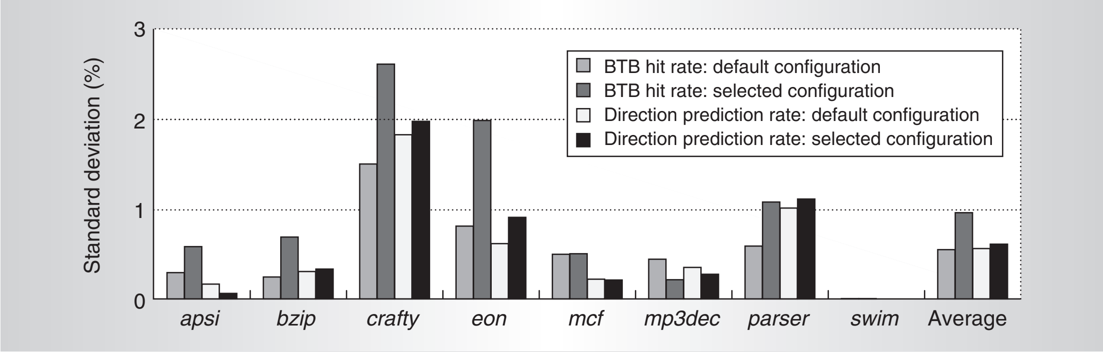

Figure 6. Standard deviation of BTB hit rate and branch direction prediction rate for module invocation, under the default configuration and the selected configuration (for saving energy).

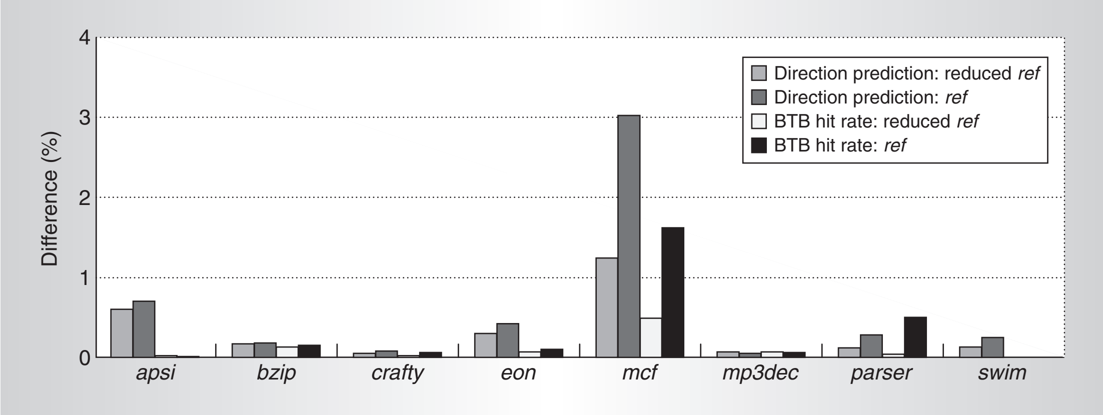

Figure 7. Weighted average of per-module difference in BTB hit rate and branch direction prediction rate between the training input and each of the two production inputs.

each module because we hypothesize that branch prediction demand is largely a function of the static code itself. To find out whether there is much variation of branch prediction demand among the dynamic instances of a module, we computed the standard deviation of the BTB hit rate and the branch direction prediction rate for all applications. Figure 6 shows the results. We computed the standard deviation of these rates among all instances of a single module and then computed the weighted average of all per-module results as the final result for the application. In general, the variation is very small. When the configuration is dynamically adjusted, the increase of standard deviation is also small.

Influence of different inputs. Our feedback-based approach gauges applications' demand using the training input and makes decisions based on that. We have already presented some evidence that this is a valid approach: The energy savings and performance impact on two sets of experiments using the training and the production inputs are similar to each other. Figure 7 shows the difference in branch prediction and BTB hit rates between different inputs. We used the same instrumented binaries to change branch prediction configurations. We executed the binaries using our default production input, the reduced version of *ref*, and the original unchanged *ref* input for 1 billion instructions after the initialization phase. We compared the per-module prediction rates to those obtained using the training input and calculated the weighted average of the absolute difference. Except for application *mcf*, the difference is quite small, suggesting that using compile-time profiling is indeed a valid approach to gauging branch prediction demand.

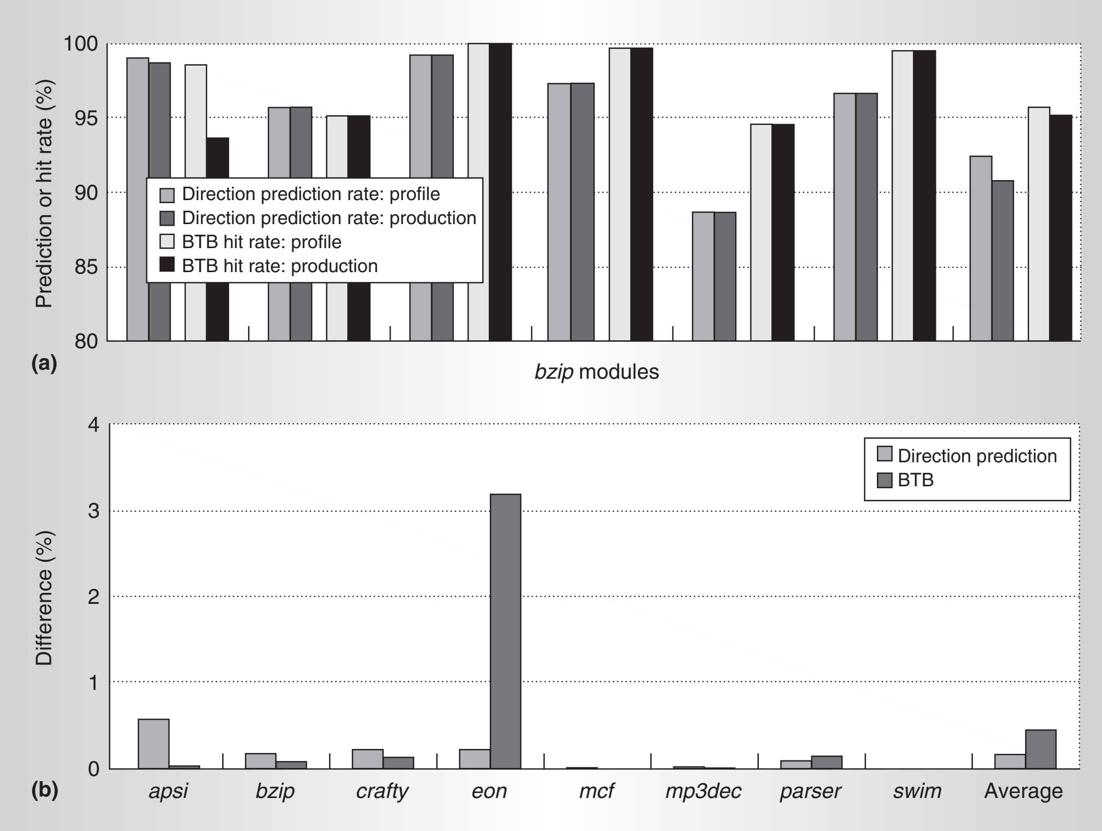

Figure 8. Projected BTB hit rate or branch direction prediction rate in profiled and actual rates in production stage: individual rates of all modules in application *bzip* (a); weighted average of *absolute* per-module difference between projected and measured rates (b).

Interference between modules. To reduce the number of profiling experiments, we made a simplifying assumption that the choice of one module's predictor configuration is independent of that of other modules. To learn whether this assumption is reasonable, we compared every module's BTB hit rate and direction prediction rate under two different cases, in which the difference was the predictor configuration for other modules. In the profiling stage, our algorithm selects for each module a configuration that we expect to achieve maximum energy savings without incurring more than a 0.5-percent execution time increase. We denote the configurations selected for module *i* as C_{M_i}. When we are testing module *i*, that module's predictor configuration is C_{M_i} in both cases. In case 1, all modules except module *i* select the full configuration. This case is the profile-time scenario. In case 2, every other module j ($j \neq i$) selects its own configuration C_{M_j}. This is the production-time scenario. To eliminate the influence of different input sets, we used the same input—the training input—in both cases.

Figure 8a shows the BTB hit rate and branch direction prediction rate of these two scenarios for application *bzip*. As the figure shows, the difference between the two cases is negligible. Figure 8b shows this experiment's results for other applications. For each module, we computed the *absolute* difference between the two measurements, and here we show the weighted average of this absolute difference.

Comparison with prediction probe detector. Finally, we compared our customized branch predictor with the prediction probe detector (PPD), a hardware-only mechanism that reduces the branch predictor's energy consumption.[13] (The "Related work" sidebar describes other research in energy-saving branch prediction.) This mechanism exploits the fact that not every cache line of instructions contains a branch, and thus, in some fetch cycles, we can avoid accessing the branch predictor. A PPD is a small cache in which each entry corresponds to a cache line in the instruction cache. Each entry contains two bits indicating whether we must access the BTB or the direction predictor when fetching the corresponding line from the instruction cache. If not, the access can be avoided or aborted. Compared with our customized branch predictor, a major advantage of the PPD is that it requires no profiling.

However, it also has the following disadvantages: First, to completely save the energy of accessing the branch predictor, we must access the PPD sequentially before accessing the predictors. This can severely limit the latency (and thus the size) of the predictors. If we access the PPD in parallel to the predictors, the mechanism can achieve only partial energy savings. For example, the sense amplification can be aborted. However, our model shows that in our environment, this energy component is less than 10 percent. Second, the PPD is itself a source of energy overhead, and it further adds to design complexity. For the predictor to determine which PPD entry to access if the instruction cache is set-associative, either the tags must be stored in the PPD, significantly increasing its size and energy, or the instruction cache must communicate the way-matching information. Either way, it takes longer to get the information from the PPD if the instruction cache is set-associative, exacerbating the first disadvantage.

We simulated an idealized PPD in which no tags were stored and access was fast enough to completely avoid any branch predictor lookup. Table 5 shows the energy savings we obtained. On average, the energy savings achieved with the customized branch predictor were even higher than those achieved with the idealized PPD. Moreover, the customized predictor achieved these savings with far less design complexity.

Related work

In addition to the prediction probe detector approach, researchers have proposed other solutions to branch prediction energy issues. Most of the proposals can work with our proposal to further reduce energy consumption.

Manne, Klauser, and Grunwald's scheme, pipeline gating, monitors the confidence of predictions for outstanding branches.[1] When the aggregated confidence is too low, instruction fetch is disabled. Whereas we try to reduce energy wasted in the branch predictor itself, these researchers try to prevent energy waste in other parts of the processor caused by ineffectual branch prediction.

Hu et al. stop powering branch predictor entries that remain unused for a long time and let them "decay."[2] This reduces the branch predictor's leakage energy. Like ours, this approach exploits the fact that many entries in modern processors' branch predictor tables are underutilized. However, this approach targets leakage energy, while ours mainly targets dynamic energy.

Other work also proposes dynamically adjusting hardware resources to reduce energy consumption while still meeting application demand. For example, Albonesi adjusts the cache configuration, Folegnani and González disable empty instruction window entries, and Bahar and Manne shut down functional units.[3-5] The concept of these approaches is similar to ours, but achieving on-demand branch prediction is a bit trickier. Although any adaptation that results in performance degradation runs the risk of increasing energy consumption (caused by fixed energy overhead per cycle), adapting the branch predictor adds an extra source of energy waste: Wrong branch predictions introduce useless instructions that are squashed later. This is not the case for these other adaptations. Moreover, to predict application demand accurately without incurring energy overhead, we adopt a feedback-based approach that exploits program behavior repetition at the module level, whereas these related proposals generally use time-based algorithms to control adaptation.

References

1. S. Manne, A. Klauser, and D. Grunwald. "Pipeline Gating: Speculation Control for Energy Reduction," *Proc. 25th Int'l Symp. Computer Architecture* (ISCA 98), IEEE CS Press, 1998, pp. 132-141.
2. Z. Hu et al., "Applying Decay Strategies to Branch Predictors for Leakage Energy Savings," Proc. *Int'l Conf. Computer Design* (ICCD 02), IEEE CS Press, 2002, pp. 442-445.
3. D.H. Albonesi, "Selective Cache Ways: On-Demand Cache Resource Allocation," *J. Instruction-Level Parallelism*, vol. 2, May 2000, http://www.jilp.org/vol2/index.html.
4. D. Folegnani and A. González, "Energy-Effective Issue Logic," *Proc. 28th Int'l Symp. Computer Architecture* (ISCA 01), IEEE CS Press, 2001, pp. 230-239.
5. R. Bahar and S. Manne, "Power and Energy Reduction via Pipeline Balancing," *Proc. 28th Int'l Symp. Computer Architecture* (ISCA 01), IEEE CS Press, 2001, pp. 218-229.

Table 5. Energy savings in the branch predictor and throughout the processor for all applications, achieved with an idealized PPD.

	Applications							
Metric	***apsi***	***bzip***	***crafty***	***eon***	***mcf***	***mp3dec***	***parser***	***swim***
$-\Delta E_{\text{Total}}$ (%)	4.69	7.43	4.71	1.96	3.44	2.87	4.22	1.84
$-\Delta E_{\text{BP}}$ (%)	75.5	63.1	40.8	47.3	33.2	58.5	40.9	81.6

We have shown that by exploiting program behavior repetition, we can implement branch predictors customized to particular program needs. This customization results in minimal degradation of prediction accuracy and performance, while achieving notable energy savings. In addition, we are considering exploiting program behavior repetition to optimize other aspects of system design. For example, we can make detailed architectural simulations more efficient by avoiding the simulation of code execution that repeats prior behavior. We are also looking at using other high-level program information to optimize resource allocation at the microarchitecture level to improve energy efficiency. MICRO

Acknowledgments

This work was supported in part by Spanish research grant TIC 2002-00750. We thank the anonymous reviewers for their helpful suggestions on improving the article.

References

1. S. Palacharla, N. Jouppi, and J. Smith, "Complexity-Effective Superscalar Processors," *Proc. 24th Int'l Symp. Computer Architecture* (ISCA 97), ACM Press, 1997, pp. 206-218.
2. A. Seznec et al., "Design Tradeoffs for the Alpha EV8 Conditional Branch Predictor," *Proc. 29th Int'l Symp. Computer Architecture* (ISCA 02), IEEE CS Press, 2002, pp. 296-306.
3. S. Yang et al., "An Integrated Circuit/Architecture Approach to Reducing Leakage in Deep-Submicron High-Performance I-Caches," *Proc. 7th Int'l Conf. High-Performance Computer Architecture* (HPCA 01), IEEE CS Press, 2001, pp. 147-157.
4. D.H. Albonesi, "Selective Cache Ways: On-Demand Cache Resource Allocation," *J. Instruction-Level Parallelism*, vol. 2, May 2000, http://www.jilp.org/vol2/index.html.
5. A. Seznec and P. Michaud, *De-aliased Hybrid Branch Predictors*, tech. report 3618, Institut National de Recherche en Informatique et en Automatique (INRIA), 1999.
6. D. Chaver et al., "Branch Prediction on Demand: An Energy-Efficient Solution," *Proc. Int'l Symp. Low-Power Electronics and Design* (ISLPED 03), ACM Press, 2003, pp. 390-395.
7. M. Huang, J. Renau, and J. Torrellas, "Positional Adaptation of Processors: Application to Energy Reduction," *Proc. 30th Int'l Symp. Computer Architecture* (ISCA 03), IEEE CS Press, 2003, pp. 157-168.
8. T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*. McGraw-Hill, 1989, pp. 333-336.
9. K. Yeager, "The MIPS R10000 Superscalar Microprocessor," *IEEE Micro*, vol. 16, no. 2, Apr. 1996, pp. 28-40.
10. V. Krishnan and J. Torrellas, "A Direct-Execution Framework for Fast and Accurate Simulation of Superscalar Processors," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques* (PACT 98), IEEE CS Press, 1998, pp. 286-293.
11. D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," *Proc. 28th Int'l Symp. Computer Architecture* (ISCA 01), IEEE CS Press, 2001, pp. 83-94.
12. A.J. KleinOsowski and D.J. Lilja, "MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research," *Computer Architecture Letters*, vol. 1, June 2002.
13. D. Parikh et al., "Power Issues Related to Branch Prediction," *Proc. 8th Int'l Symp. High-Performance Computer Architecture* (HPCA 02), IEEE CS Press, 2002, pp. 233-244.

Michael C. Huang is an assistant professor in the Department of Electrical and Computer Engineering and the Department of Computer Science at the University of Rochester, New York. His research interests include computer system architecture and processor microarchitecture, with emphases on adaptive architecture, power-aware design, and system optimization. Huang has a PhD in computer science from the University of Illinois at Urbana-Champaign. He is a member of the IEEE and the ACM.

Daniel Chaver is pursuing a PhD in the Computer Architecture Department of the Universidad Complutense de Madrid (UCM). His research interests include energy-aware architectures and code optimization for modern microprocessors. Chaver has an MS in physics from the Universidad de Santiago de Compostela and an MS in electrical and computer engineering from UCM. He is a student member of the IEEE and the Computer Society.

Luis Piñuel is an assistant professor in the UCM Computer Architecture Department. His research interests include microprocessor architecture with emphasis on value prediction, power-aware processors, and code optimization. Piñuel has MS and PhD degrees in computer science from UCM. He is a member of the IEEE and the Computer Society.

Manuel Prieto is an associate professor in the UCM Computer Architecture Department. His research interests include code optimization, performance, power, and memory issues in superscalar and embedded processors. Prieto has an MS in computational physics and a PhD in computer science, both from UCM. He is a member of the IEEE and the Computer Society.

Francisco Tirado is a professor in the UCM Computer Architecture Department. His research interests include energy-aware processors, dynamic optimization, and parallel architectures. Tirado has MS and PhD degrees in physics from UCM. He is a senior member of the IEEE, a member of the ACM, and adviser to the Spanish National Agency for Research and Development.

Direct questions and comments about this article to Michael C. Huang, Dept. of Electrical and Computer Eng., University of Rochester, PO Box 270231, Rochester, NY 14627-0231; michael.huang@rochester.edu.