Smart Cache: A Self Adaptive Cache Architecture for Energy Efficiency

Karthik T. Sundararajan
School of Informatics
University of Edinburgh
Email: tskarthik@ed.ac.uk

Timothy M. Jones
Computer Laboratory
University of Cambridge
Email: timothy.jones@cl.cam.ac.uk

Nigel Topham
School of Informatics
University of Edinburgh
Email: npt@staffmail.ed.ac.uk

***Abstract*—The demand for low-power embedded systems requires designers to tune processor parameters to avoid excessive energy wastage. Tuning on a per-application or per-application-phase basis allows a greater saving in energy consumption without a noticeable degradation in performance. On-chip caches often consume a significant fraction of the total energy budget and are therefore prime candidates for adaptation.**

Fixed-configuration caches must be designed to deliver low average memory access times across a wide range of potential applications. However, this can lead to excessive energy consumption for applications that do not require the full capacity or associativity of the cache at all times. Furthermore, in systems where the clock period is constrained by the access times of level-1 caches, the clock frequency for all applications is effectively limited by the cache requirements of the most demanding phase within the most demanding application. This results in both performance and energy efficiency that represents the lowest common denominator across the applications.

In this paper we present a Set and way Management cache Architecture for Run-Time reconfiguration (Smart cache), a cache architecture that allows reconfiguration in both its size and associativity. Results show the energy-delay of the Smart cache is on average 14% better than state-of-the-art cache reconfiguration architectures. We then leverage the flexibility provided by our cache to dynamically reconfigure the hierarchy as a program runs. We develop a decision tree based machine learning model to control the adaptation and automatically reconfigure the cache to the best configuration. Results show an average reduction in energy-delay product of 17% in the data cache (just 1% away from an oracle result) and 34% in the level-2 cache (just 5% away from an oracle), with an overall performance degradation of less than 2% compared with a baseline statically-configured cache.

I. Introduction

The power dissipation of modern microprocessors is a primary design constraint across all processing domains, from embedded devices to high performance chips. Shrinking feature sizes and increasing numbers of transistors packed into a single die only exacerbates this issue. Schemes are urgently required to tackle power dissipation, yet still deliver high performance from the system.

Cache memories contain a large number of transistors and consume a large amount of energy. For instance, 60% of the StrongARM's area is devoted to caches [1]. For this reason many processors, particularly intellectual property cores, allow the configuration of the caches to be determined at design time, according to the requirements of the target applications. Customization of cache parameters may be static or dynamic; in a static approach the designer sets the cache parameters before synthesis, whereas in a dynamic scheme the cache parameters can be modified within a certain range at run-time.

When cache parameters are determined statically, a single configuration is chosen by the designer to trade-off performance against energy consumption. Static configurations require less on-chip logic, validation and testing than performing dynamic reconfiguration. However, they do not have the ability to react to changes in cache requirements both across programs and within the same application. In order to achieve optimum energy efficiency, cache parameters should be reconfigured at run-time in response to the changing requirements of the running application.

Dynamic cache reconfiguration is not a new topic, having been previously studied by a variety of researchers [2], [3], [4], [5], [6], [7], [8], [9], [10]. These schemes monitor the miss ratio at run-time, reconfiguring the cache whenever it reaches a certain threshold value. However, they are limited in the amount of flexibility they provide — either performing set-only or way-only reconfiguration — or they consult larger sub-banks on each access than are actually required. Furthermore, relying solely on the miss ratio to determine the correct time to reconfigure does not always give a good indication of the changing requirements of the application.

This paper makes the following contributions:

- We first propose a configurable cache architecture that allows reconfiguration of both the size and associativity of each cache, providing maximum flexibility to the application. We compare our approach, called the Smart cache, against state-of-the-art cache reconfiguration techniques and show that our scheme's energy-delay product is on average 14% better than these prior works.
- To demonstrate the performance of our scheme we develop a decision tree model that monitors the behavior of each cache and dynamically reconfigures in response to changing application requirements. We demonstrate that our approach causes negligible performance loss, yet achieves an energy-delay product of 0.83 and 0.66 in the data cache and level-2 cache respectively.

The rest of this paper is structured as follows. Section II describes related work and the importance of our cache architecture. Section III describes our Smart cache and section IV

presents our decision tree model with the features of cache behavior that it monitors. It also discusses the power and performance overheads of our approach. Section V describes our experimental set-up and section VI presents our results. Finally, section VII concludes.

II. Related Work

This section describes the existing state-of-the-art reconfigurable cache architectures and explains the need for and importance of our work.

Reconfigurable caches are not new. Several researchers have investigated configurable cache designs that vary parameters such as the size, line size and associativity. The state-of-the-art reconfigurable cache architectures can be grouped into the following categories.

A. Set-Only Reconfiguration

In a set-only cache, the cache size is increased and decreased by enabling or disabling one or more sets respectively [11]. At smaller cache sizes, unused sets can be turned off to reduce the static energy consumption [8]. The miss ratio was used by Yang et al. [11] to guide cache reconfiguration, varying the size by masking index bits through a shifting operation. This allowed them to alter the cache size one step at a time. Our approach, in contrast, allows us to alter both size and associativity and reconfigure to any configuration in just one step.

B. Way-Only Reconfiguration

Albonesi [3] proposed a cache design that can vary size and associativity by enabling or disabling cache ways, saving dynamic power when using less resources. This is a coarse-grained reconfiguration approach that may increase capacity and conflict misses [11].

Zhang et al. [12] proposed way-concatenation to reduce dynamic power by accessing fewer ways, depending on the associativity. This was performed once, before an application started execution. They also used way-shutdown to decrease cache size by turning-off unused ways using the Gated-V_{dd} method [8]. However, they did not address the changes required to the control signals when adding in way-shutdown. Later, Ross et al. [5] described an extension to enable dynamic cache reconfiguration. However, they do not describe the control signals required to combine way-concatenation with way-shutdown. Furthermore, this requires flushing of dirty data to the next level cache when increasing associativity for a fixed cache size, something that our Smart cache avoids.

Orthogonal to this, way-prediction schemes can be used to reduce cache power by only accessing the ways that contain the required data [13], [14].

C. Set-and-Way Reconfiguration

Yang et al. [9] combined configurable set [8] and way [3] architectures to offer a hybrid cache that gives flexibility in terms of size and associativity. Increasing associativity by adding ways but keeping the cache size fixed results in a copying back of previously stored data. This shows the need for our cache architecture that supports dynamic reconfiguration without incurring extra cycles for copying back information, while increasing the associativity. Furthermore, our Smart cache is complementary to heterogeneous way-sizes [2], concatenating lines [15] and wide-tag partitioning [4].

D. Other Approaches

Focusing on leakage energy saving, Flautner et al. [16] proposed drowsy caching that preserves state when in a low-power mode. Kaxiras et al. [17] developed cache decay which is a state-destroying low power scheme. However, neither of these techniques reconfigures the cache — they simply place lines in a low power mode. Their ability to save energy relies on the correct selection of the interval that lines are placed in a low power mode. Moreover, they reduce only static and not dynamic power, whereas our scheme reduces both static and dynamic energy significantly.

Powell et al. [8] proposed a gated-Vdd (non-state preserving) technique to reconfigure the cache and turn off unused cache lines. Meng et al. [18] explored the upper limits of reducing leakage power by combining both drowsy and gated-Vdd techniques. However, this work is only a theoretical upper bound since it assumes the existence of an ideal pre-fetcher which is impossible to provide in practice.

Micro-architecture design space exploration (including for caches) has been studied by several researchers using linear regression models [19], artificial neural networks [20], [21], [22], [23], radial basis functions [24], and spline functions [25], [26], [27]. However, these papers assume indefinite hardware resource availability and also none of these papers addresses the question of how to achieve these benefits. Our paper presents a flexible cache architecture that can be reconfigured on-line to enable power savings.

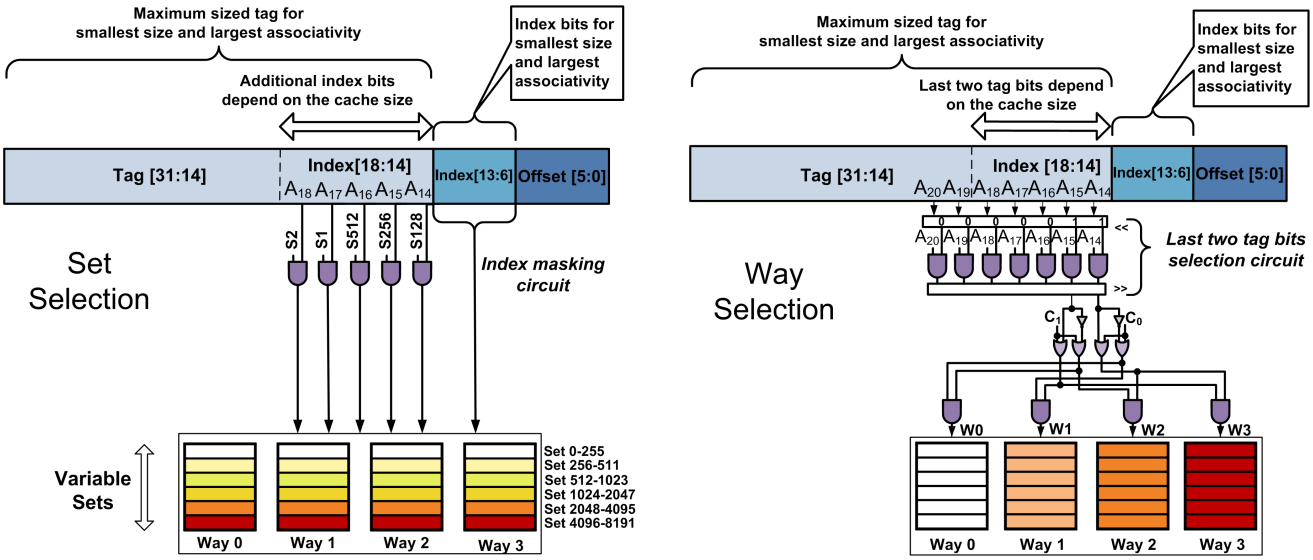
III. The Smart Cache Architecture

This section describes the Smart architecture that we use for each cache within the system. Figure 1 shows how each address is mapped into our cache for a 2MB level-2 cache. There are two complementary circuits used in parallel that perform the mapping, allowing the address to be routed to the correct set and way.

As the cache size and associativity varies, so does the number of bits needed for the tags. In our architecture, we store the maximum sized tag for each line (i.e., for the smallest cache and largest associativity). We now describe how the address is routed to the sets and ways, then discuss the overheads of our architecture.

A. Set Selection

We group the sets in each bank by augmenting the cache with size selection bits that determine the sets that are enabled. These are then *AND*ed with bits from the index to determine the sets to access. In the 2MB level-2 cache shown in figure 1 we have size selection bits S128, S256, S512, S1, and S2 representing cache sizes of 128KB (sets 0-511), 256KB (sets 0-1023), 512KB (sets 0-2047), 1MB (sets 0-4095) and 2MB



(a) Set Selection (b) Way Selection

Fig. 1. Organization of the Smart cache architecture. Varying the cache size (through the set selection circuits) is performed in parallel with altering the associativity (through the way selection logic).

(sets 0-8191) respectively. A 64KB cache (sets 0-255) is always enabled even when all size selection bits are 0. The size selection bits could be set via a hardware scheme, or exposed to the software.

B. Way Selection

In order to control the associativity of the cache, we augment the cache with a way selection circuit. This uses the last two tag bits and the size selection bits to route accesses to the ways that are enabled. Since the cache size can vary, the size selection bits are required to correctly identify the last two tag bits. In figure 1, for a small cache they will be bits [15:14] and for a large cache bits [20:19]. Two control bits (C_0 and C_1) determine the ways that will be eventually accessed. For one-way associativity, any one of the way selection control signals W0, W1, W2 or W3 will be active. For two-way associativity, any two of the way selection control signals will be active. Finally, for four-way associativity, all way selection control signals are active. Table I shows how the control and tag bits map to the ways that are enabled.

C. Example Cache Access

Figure 2 shows how the control and tag bits map to the ways that are enabled in our Smart cache. As an example of how the set and way selection circuits work in tandem, consider a 512KB, two-way associative cache. In this scenario, size selection bits S128, S256 and S512 are set to 1, all others are set to 0. For the control bits, C_0 is set to 0 and C_1 is set to 1.

The address is routed to cache banks after passing through the selection circuits. For all the cache configurations, the tag bits [31:14] are used. Depending on the cache size, the way selection circuit selects two tag bits, in this example it uses bits [18:17] which corresponds to a cache size of 512KB.

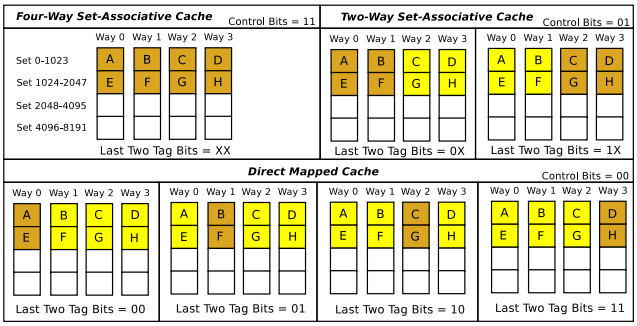


Fig. 2. Working of the Smart cache. Brown, yellow and white regions show accessed, unaccessed and disabled sets respectively. Control bits determine the associativity and the last two tag bits determine the ways.

Assuming that they are both 1, then ways W2 and W3 are accessed. In the set selection circuit, bits [16:6] are also passed through with the index to select the correct lines from sets 0-2047. All other sets are turned off for static power saving.

D. Overheads

The way selection circuit does not appear in the cache's critical path because it can operate in parallel with the tag and data array address decoders [12]. The set selection circuit can be folded into the decoders to avoid any delay in calculating the index bits [8]. Therefore there is no increase in the cycle time for accessing the cache using our approach. However, after reconfiguring the cache to a smaller size, we turn off unused sets and ways, destroying their contents. Therefore we must flush any dirty lines back to the next level in the memory hierarchy. In all our simulations we add the flushing cost that includes both power and performance costs for copying back the dirty lines. Power and performance overheads of reconfiguration are discussed in detail in section IV-C.

TABLE I
MAPPING OF TAG AND CONTROL BITS TO THE ACTIVE WAYS.

Associativity	Control Bits C_0	Control Bits C_1	Last Two Tag Bits	Active Way Signals
One Way	0	0	00	W0
			01	W1
			10	W2
			11	W3
Two Way	0	1	0X	W0, W1
			1X	W2, W3
Four Way	1	1	XX	W0, W1, W2, W3

We have calculated the area overheads of the cache as 0.5% over the baseline. This is due to the extra control circuitry required to perform set selection, way selection and reconfiguration. This value has been obtained from a version of Cacti-5.3 [28] that has been modified to support our new circuitry.

E. Relation to Prior Work

There are several key differences between our Smart cache and state-of-the-art reconfiguration techniques. In our approach the associativity and size are varied in parallel by using the way control signals and the size control registers. The Smart cache organizes ways at set boundaries, which avoids flushing data back to memory when increasing the associativity but keeping the cache size fixed. This addresses the shortcomings from previous techniques [12], allowing dynamic reconfiguration of the cache. In addition to this, the Smart cache offers 3x more cache configurations than the set-only [11] and hybrid [12] schemes, which combine way-concatenation with way-shutdown.

IV. CONTROLLING RECONFIGURATION

Having developed our cache architecture, this section now describes our method for dynamic reconfiguration. We monitor the cache behavior by collecting statistics about the cache usage over a fixed interval size. These are then fed into a decision tree that computes the required cache size and associativity for the next interval. We first present the statistics used to characterize cache behavior, then describe the decision tree itself.

A. Cache Behavior Characterization

In order to determine the best cache configuration to use for each program interval, we monitor the cache behavior by gathering statistics about cache usage. These allow us to accurately determine when the cache size or associativity needs to be altered. We gather two types of statistic: stack distance and dead set count.

1) Stack Distance: The stack distance [29] shows the position in a set's LRU chain that each access occurs in. This gives an approximation of the required associativity of the cache: if all accesses are in the MRU position, then the associativity can be reduced; if many accesses are in the LRU position or miss then the cache could benefit from higher associativity. We maintain a counter for each position in the LRU chain for the whole cache to enable us to gather this information.

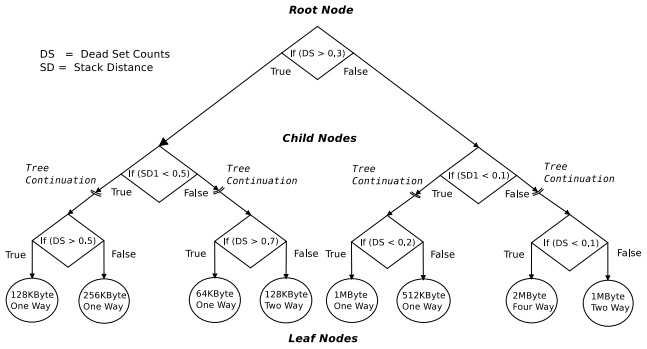


Fig. 3. Example decision tree structure.

2) Dead Set Counts: We define a dead set as one that is not accessed during a clearing interval (10K committed instructions). A large number of dead sets indicates that the cache could function adequately with a smaller number of sets (i.e., a smaller size). To monitor this we add a 2-bit saturating counter to each set, clear them at the start of each clearing interval and increment on each access. At the end of the each clearing interval these counters are used to compute the total number of sets that have been accessed less than three times, and are averaged over phase interval (10M committed instructions). This is a simple statistic, but it accurately identifies dead sets.

B. Decision Tree Model

Any form of dynamic hardware reconfiguration requires a decision-making process, driven by run-time measurements. This could be derived intuitively, but would then be open to the criticism that the model is trained specifically for the selected benchmarks. We take a different approach, choosing machine learning to train a decision tree model, which clearly separates training and experimental data. An example is shown in figure 3. At each node in the tree, any one of the collected statistics is compared to a threshold value and, depending on the outcome control passes either left or right to the corresponding child node. We used decision trees to control reconfiguration because they can be easily implemented in hardware using a look-up table.

Assuming a dead set count (DS) of 0.6 and stack distance (SD) of 0.3, we can follow the example tree in figure 3 to find the required configuration. The first comparison is performed in the root node where DS is compared to the threshold value of 0.3. We therefore take the edge labeled *True* and proceed down the left to the next node. This compares SD with 0.5, so we take the edge labeled *False*. The final comparison considers whether DS is greater than 0.7, which is also *False*. We therefore arrive at the node containing the desired configuration, which is a 128KB cache with 2-way associativity.

In order to determine the thresholds at each node, the decision tree needs to be trained using examples of good

configurations from different programs. Training consists of finding thresholds that minimize the partition variance at each node. To do this we ran each training program on all cache configurations and gathered the characterization statistics every interval. We defined good configurations as those that have an energy-delay lower than the baseline, with a maximum slowdown of 2% from the baseline across each interval. We then used leave-one-out cross-validation to train our decision tree using this data. This is a standard machine learning methodology and ensures the model is never trained on the benchmark it is tested on.

C. Overheads of Reconfiguration

There are two types of overhead that our dynamic reconfiguration scheme incurs. The first is power consumption and the second is performance.

a) Power Consumption: We have calculated the power consumption of our statistics gathering logic for each cache in the processor and the models used to drive reconfiguration. These have been incorporated into our simulator and the overheads included in all results. The energy overheads of the statistics gathering logic are 0.01% of the baseline cache energy. The overhead of the decision tree model is 1% of the baseline cache energy consumption.

b) Performance: Traversing the decision tree to find the best cache configuration for the next interval takes several cycles. However, this is small in comparison to the time taken to run each interval. By halting our characterization shortly before the end of the interval, we can overlap the decision tree traversal with the execution of the end of the interval, hiding its latency. The performance overheads in actually performing reconfiguration of each cache are described in Section III and are included in all of our results.

Altering the cache size or associativity may require dirty data to be written back to lower-level memory. When cache size is reduced by turning-off sets or ways, requires dirty lines present in the future turned-off region to be written back to next lower level. When cache size is increased, blocks may map to different sets. This incurs extra misses for the first accesses to the new location and also requires dirty lines to be flushed back to the next level before increasing the size.

These reconfigurations incur extra cycles to copy back dirty lines to lower levels of memory, which incurs extra performance and energy costs. However our experimental results show that on average reconfiguration is required once in every 10 intervals, or once every 100 million instructions. Thus, costs associated with this can be quantified by not varying cache size and associativity very often. The performance and energy costs of flushing these cache lines are included in all our results and, since reconfiguration is performed so infrequently, the overheads are small.

V. Experimental Setup

This section describes the simulator and benchmarks used to evaluate our cache reconfiguration approach.

TABLE II
PROCESSOR CONFIGURATION.

Parameter	Configurations
Decode,Issue,Commit Width	4 ,4, 4
Register Update Unit Size	80
Load Store Queue Size	40
Instruction Cache Size	1 → 32 KBytes
Instruction Cache Associativity	1 → 4
Instruction Cache Line size	32 Bytes
Data Cache Size	1 → 32 KBytes
Data Cache Associativity	1 → 4
Data Cache Line size	32 Bytes
Level-2 Cache Size	64 → 2048 KBytes
Level-2 Cache Associativity	1 → 8
Level-2 Cache Line size	64 Bytes
Level-2 Cache Latency	6 Cycles
Memory access bus width	8 Bytes
Main-Memory Latency	97 Cycles
Technology	70nm

We implemented cache reconfiguration in the HotLeakage simulator [30]. We updated the underlying power models to use a more recent version of Cacti-5.3 [28] that has been modified to support our new circuitry for 70nm process technology. We also altered the simulator to include the power and performance overheads of reconfiguring each cache, as previously described in Sections III and IV. Table II shows the configuration of our Alpha out-of-order superscalar, whose cache configurations are similar to an Intel Core 2 processor.

To evaluate our technique we used the SPEC CPU 2000 benchmark suites [31] as workloads, compiled with the highest optimization level. We used the *reference* inputs for running each application. Due to simulation time constraints and to maintain the continuity of cache behavior, we ran each workload from its start to 60 billion instructions. This ensures that we cover the majority of each benchmark's behavior.

In our simulations we assumed a phase interval of 10 million instructions. We chose this after a characterization of the benchmarks using sampling intervals of 10K, 100K, 1M and 10M instructions. This this also is a value commonly used by other researchers [32]. To gather data to train our decision tree model we ran each CPU 2000 benchmark on each cache configuration, gathering cache characterization statistics every interval. With 23 applications, 3 caches and 18 configurations for each, this totals 1,242 simulations. We then used leave-one-out cross-validation, a standard machine learning evaluation methodology to evaluate our scheme, as described in Section IV.

We have used WEKA to analyze our training data-sets using data-mining algorithms [33]. WEKA comprises data classification, regression, clustering, association rules and visualization. It analyzes, pre-processes and selects the key features from the training data-set and applies classification algorithms on the selected features.

VI. Results

This section evaluates our Smart cache approach to dynamic cache reconfiguration. We first perform a comparison with prior cache architectures on static configurations of the level-2

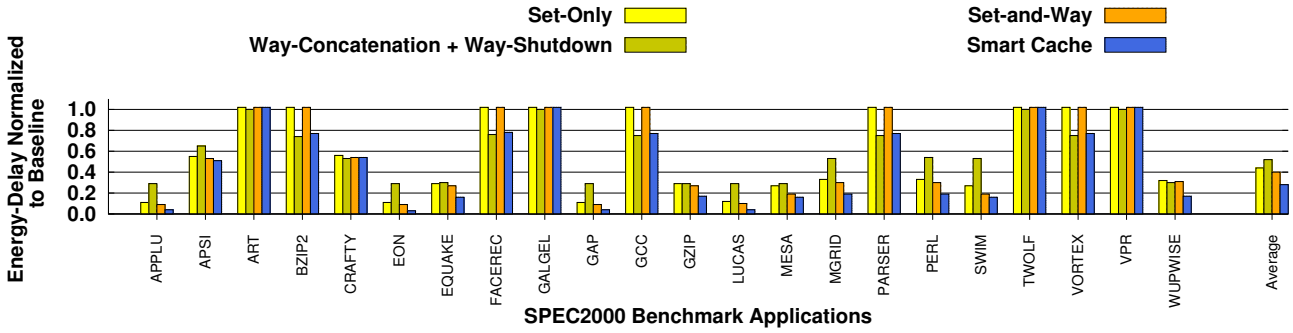


Fig. 4. Energy-delay values for different cache architectures running on the baseline level-2 cache configuration.

cache. We then show the effects of dynamic reconfiguration using our architecture and decision tree model on each cache individually and a combined scheme for all caches at once.

In later graphs we show the performance of our approach and the energy-delay product achieved for the whole cache hierarchy, taking into account reconfiguration and flushing costs. In addition to this, we show two comparison techniques. The first is the best static configuration of the cache, which corresponds to the configuration that has the lowest energy-delay and a maximum 2% performance loss across all benchmarks, with no dynamic reconfiguration. These are the same criteria used to select good configurations to train our decision tree. The second approach is an ideal oracle which knows in advance the best configuration for each interval and incurs no overheads in dynamic reconfiguration. Although unrealistic in practice, this represents the lower bound on achievable energy-delay for any technique.

All energy-delay results are normalized to the baseline architecture which has an energy-delay value of 1.0. This is a processor where each cache is configured to its largest size and highest level of associativity.

A. Comparison With Prior Work

Our first evaluation compares our cache architecture with prior state-of-the-art designs. Figure 4 shows set-only cache [8], which can increase/decrease cache size; Way Concatenation cache [12], which concatenates one or more ways to get the desired associativity and also uses way-shutdown to turns-off the unused ways to reduce leakage power. Set and Way cache [9], which incorporates both [3], [8] schemes; and our Smart Cache. We show the energy-delay product achieved when running each benchmark on the best static configurations for that application for each cache architecture. The best static configuration is the one that has the lowest energy-delay and a maximum 2% performance loss, from all the possible configurations.

For benchmarks such as *art*, *bzip2*, *facerec*, *galgel*, *gcc*, *parser*, *twolf*, *vortex* and *vpr* the best static configuration is 2MB with eight-way associativity so there are no energy savings achievable for any cache architecture. The set-only, set-and-way and Smart approaches consume around 1.7% more energy compared to way-concatenation, because the latter does not use extra tag-bits that other architectures require to change the cache size. For some benchmarks, like *bzip2*, *facerec*, *gcc*, *parser* and *vortex*, the set-only and set-and-way approaches do not do well compared to the way-concatenation and Smart caches. The reason for this is that these benchmarks require a 2MB cache with two-way associativity which is only offered by way-concatenation and Smart cache. For these architectures, dynamic energy is reduced by accessing fewer ways, which is not possible in the set-only and set-and-way caches.

For others benchmarks, such as *applu*, *eon*, *gap* and *lucas* significant energy-delay reductions can be achieved. This is due to our approach accessing fewer sets and ways as compared to the set-only and set-and-way approaches for lower associativity. It can also be seen that no single approach can provide good energy-delay values for all applications.

Overall, the average level-2 cache energy-delay achieved by our approach is 0.28, which is 14% better than set-only and set-and-way approaches and 25% better than the way-concatenation with way-shutdown approach. This clearly demonstrates the benefits of using our architecture for cache reconfiguration. The next section now harnesses this flexibility to dynamically reconfigure level-2 cache to obtain further power savings.

B. Dynamic Cache Reconfiguration

This section evaluates our Smart cache architecture along with our decision tree model for dynamic reconfiguration of each cache in the hierarchy individually. In the following subsections, when reconfiguring the instruction cache, the energy spent in the baseline data and level-2 caches are added to the energy of the instruction cache. This is done to ease the comparison between the cache hierarchies. The same has also been employed for reconfiguring data and level-2 caches.

1) Instruction Cache: Figure 5 shows the performance and energy-delay of three different schemes when reconfiguring the instruction cache alone. As previously described, the first represents the best static configuration of the instruction cache across all benchmarks and the second is the oracle. The bars labeled *Smart cache* show the results from using our decision tree model along with our Smart architecture. The black circles show the performance achieved by Smart scheme,

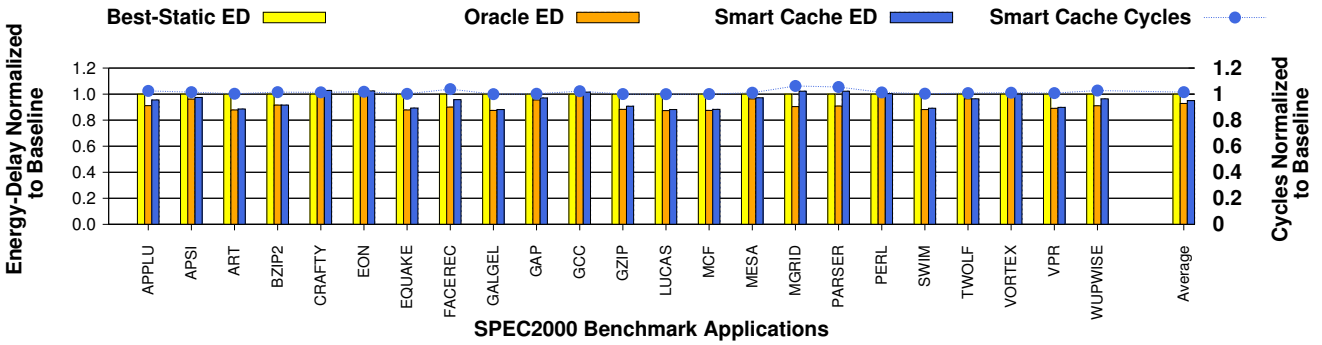


Fig. 5. Performance and energy-delay characteristics of the instruction cache, while maintaining the data and level-2 caches at the baseline configuration. This chart shows the percentage reduction in total energy-delay achieved by reconfiguring only the instruction cache.

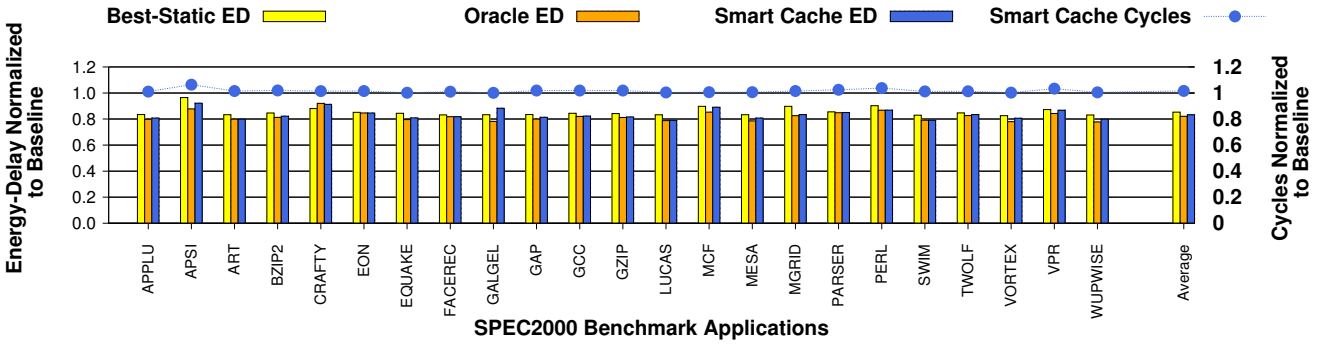


Fig. 6. Performance and energy-delay characteristics of the data cache, while maintaining the instruction and level-2 caches at the baseline configuration. This chart shows the percentage reduction in total energy-delay achieved by reconfiguring only the data cache.

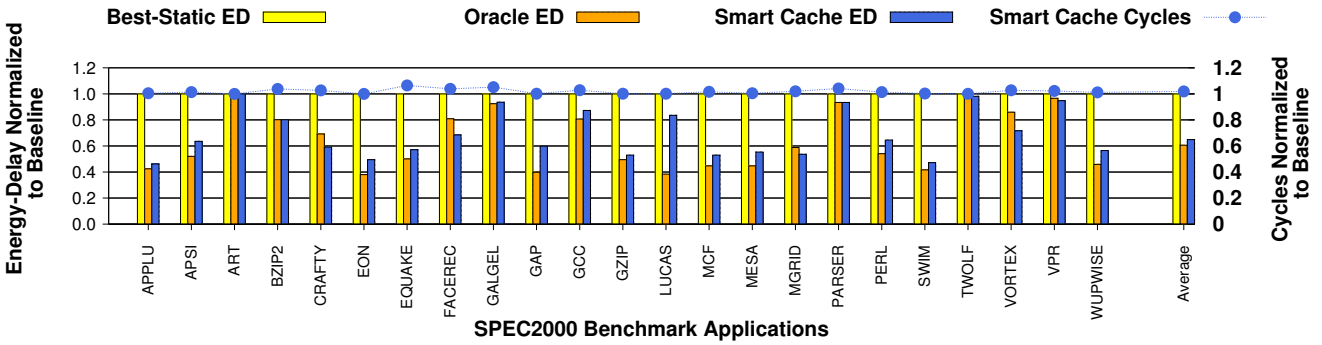


Fig. 7. Performance and energy-delay characteristics of the level-2 cache, while maintaining the instruction and data caches at the baseline configuration. This chart shows the percentage reduction in total energy-delay achieved by reconfiguring only the level-2 cache.

normalized to the baseline performance. The oracle and best static approaches never incur more than 2% performance loss, so their performance results have been excluded. For the instruction cache, the best static configuration is actually the 32KB cache with 4-way associativity (i.e., the baseline).

On average the energy-delay of our scheme is close to the theoretical maximum limit achieved by the oracle, with the difference being 2.2%. However, we lose 6% and 5% performance on *mgrid* and *parser* respectively, due to our decision tree model predicting too small a cache configuration at the phase transitions. This is because the transition phase cache statistics for *mgrid* are similar to those from *applu* and *apsi* which need caches that are 2KB large, whereas *mgrid* requires a cache of 8KB. For the other applications, our decision tree model is effective at determining the correct cache configuration to use. Therefore, on average we incur a performance loss of just 1.5% compared to the baseline, but achieve an energy-delay value of 0.95. A small performance loss such as this is expected since we chose to bound performance losses to 2% of the baseline when identifying good configurations, as described in Section IV-B.

2) Data Cache: Turning our attention to the data cache, shown in figure 6, we see that there is greater improvement to be gained than can be achieved from the instruction cache.

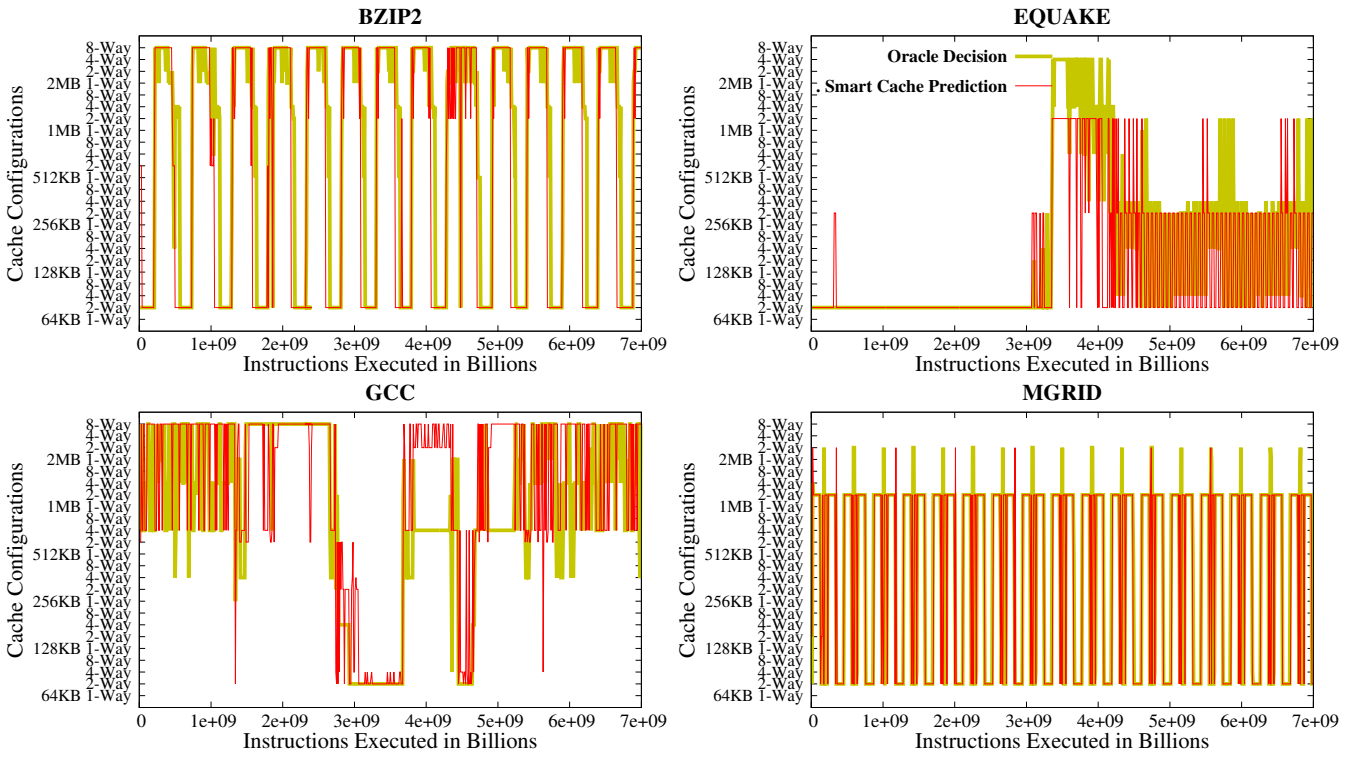


Fig. 8. Dynamic cache configuration traces, illustrating the correspondence between the Oracle and the Smart cache reconfiguration behavior. The y-axis shows different cache configurations and the x-axis shows the time interval of instructions executed.

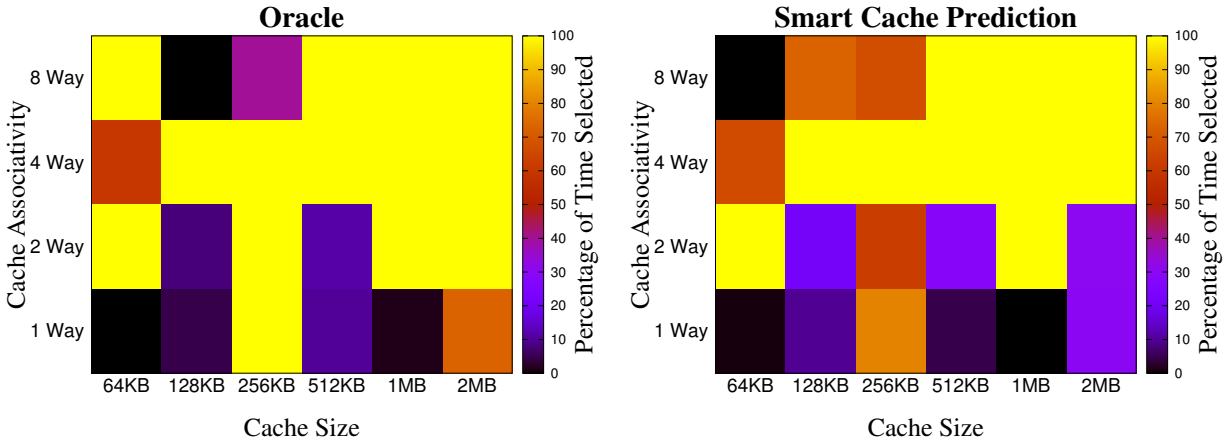


Fig. 9. Heatmaps showing the distribution of level-2 configurations required by the oracle and Smart cache across all SPEC CPU 2000 applications.

Here we incur a similar performance loss of 1.6% on average, rising to 6.3% for *apsi*. This is again due to inaccuracies in our decision tree model that infers a smaller cache configuration during phase transitions for this application than is actually required.

Considering the energy-delay, figure 6 shows that we again achieve results close to the oracle. The difference here is 1.1%. On average we achieve an energy-delay value of 0.83. This is consistent across applications with *art*, *lucas*, *swim* and *wupwise* achieving values less than 0.8.

3) Level-2 Cache: We now consider the final cache in the hierarchy, which is the unified level-2 cache. Figure 7 shows the results of dynamically reconfiguring the level-2 cache. The best static cache configuration, across all benchmarks, is actually the baseline 2MB 8-way cache, so without dynamic reconfiguration, no energy savings are possible. Figure 8 then shows how the cache configurations selected by the Smart cache compare over time with the oracle's selection.

a) Performance And ED: These results show that our Smart cache is able to obtain significant energy-delay improvements with only minimal performance overheads. The average performance loss for our approach is 1.8%, which is within our target value that was used to determine good cache configurations. Applications like *equake* and *galgel* incur 6.4%

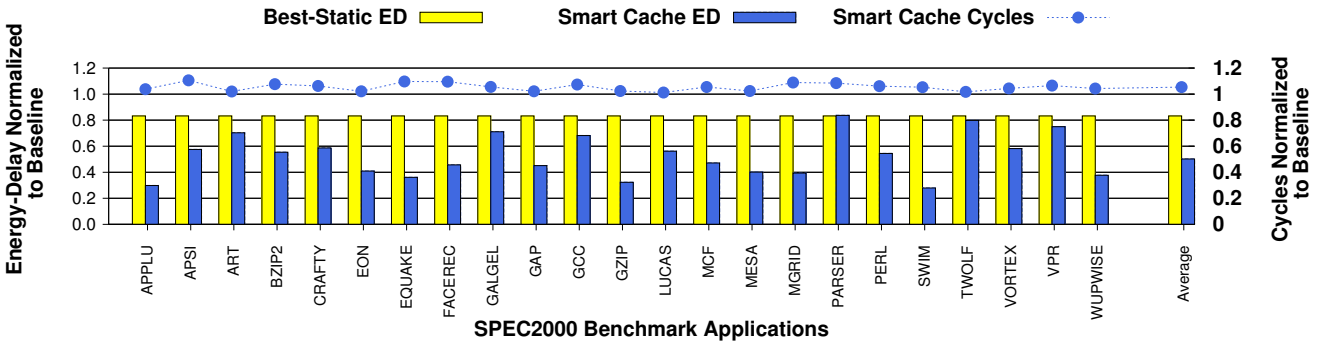


Fig. 10. Combined performance and energy-delay characteristic of all three caches within the cache hierarchy, showing an overall reduction in energy-delay of 50%

and 5.2% performance losses respectively which is primarily due to choosing smaller caches during the transition phase. In *crafty*, *facerec*, *mgrid*, *vortex* and *vpr*, our Smart cache performance is slightly over the actual limit of 2%, whereas the oracle and best-static schemes are within the performance limit. The reason being that for a few small phases in the application, our model predicts a smaller cache size than required and hence it incurs extra performance losses, whilst also reducing energy-delay.

In terms of the energy-delay product, *eon* achieves 0.49 with no performance loss. This is due to *eon* mainly requiring a cache of 128KB. For *lucas*, our approach achieves a value of 0.83, whereas the oracle scheme is at 0.38. The cache statistics for *lucas* are similar to *parser* and *vpr*, when it is actually more similar to *applu*, *eon*, *gap* and *swim* in terms of cache size requirements.

In terms of the average energy-delay product, we achieve a value of 0.66 — a significant reduction compared to the best static approach. This shows the benefits of dynamic reconfiguration of the level-2 cache using our approach.

b) Configurations Selected: To consider how these savings are achieved, figure 8 shows how the predictions made by our Smart cache vary as an application runs. Also shown for comparison is the oracle approach. Due to space limitations we have only shown the results from four representative benchmarks.

In *bzip2*, there is a regular pattern of configurations required, alternating between a 64KB, 2-way cache and a 2MB configuration. It is clear from the diagram that our approach accurately tracks the oracle and leads to the savings shown in figure 7.

The next two benchmarks (*equake* and *gcc*) have irregular patterns. For the majority of the time, the Smart cache can accurately determine the correct configuration to use. However, sometimes it predicts too small a cache size (in *equake*), leading to performance losses or too large a configuration (in *gcc*), leading to higher ED values than are optimal.

The final benchmark is *mgrid* which is interesting because we obtain a lower ED value than the oracle. As can be seen in figure 8, this is due to the Smart cache accurately reconfiguring the cache as the oracle scheme does, but occasionally using a smaller size which leads to negligible performance losses but increased energy savings. Overall, figure 8 shows that the Smart cache is able to track the configurations chosen by the unrealistic oracle scheme.

A summary of the configurations required by both the oracle and our Smart cache can be seen in figure 9. We present the results as a heat map, where darker blocks correspond to more frequently chosen configurations. These figures are averaged across all SPEC CPU 2000 applications.

The most frequently-used configuration is the 2MB, 4-way cache. In contrast, a direct-mapped cache is rarely chosen by either scheme, and nor is the smallest cache size of 64KB, apart from the 2-way configuration that is useful for certain benchmarks, as seen in figure 8. From these heat maps it is clear that the Smart cache's predictions are closely correlated to the configurations chosen by the oracle, providing further evidence of the accuracy of our approach.

C. Cache Hierarchy Reconfiguration

Having shown the benefits of reconfiguring each cache individually, this section evaluates the effects of reconfiguring each cache in the hierarchy at the same time. Figure 10 shows the results. We show the best static configuration and our approach only. We do not have results for the oracle scheme because this would require a complete evaluation of the design space (i.e., 128,304 simulations) which is impractical within our current setup.

As figure 10 shows, applications such as *applu*, *art*, *eon*, *gap*, *gzip*, *lucas*, *mesa*, *twolf* and *wupwise* incur small performance losses of under 4%. However, other benchmarks experience larger losses, leading to an average performance loss of 5.3%.

On the other hand, there are significant improvements in the energy-delay values achieved. Our approach is always better than the best static configuration with *swim* achieving a value of 0.27 and *applu* achieving 0.29. The reason behind the decrease in performance when all caches change simultaneously is due to the selection of inappropriate cache configurations during transition phases. This can be observed by comparing figure 10 against other three individual cache changing schemes shown in figures 5 to 7.

For example, in *apsi*, individually changing instruction and

level-2 caches incurs less than 2% performance loss. However, when changing all caches at once, our model mistakenly selects too small a size for the data cache and this influences predictions made by for level-2 cache, increasing overall performance losses. A similar effect can be seen in *bzip2*, *equake*, *facerec*, *mgrid* and *parser*. Since we start our experiments from the beginning of each application and execute 60 billion instructions without using any profiled phase information, many small transition phases are encountered in our experiments. These transition phase boundaries could be easily identified by a phase detector [32], [34], which would allow us to vary the interval length and reconfigure more accurately. However, on average, our Smart cache approach achieves an energy-delay of 0.50, almost half that of the best static scheme.

D. Summary

This section has presented the results from our Smart cache approach. It is clear from figures 5 and 6 that reconfiguring the instruction and data caches does not bring many benefits. This is because of their small sizes in comparison to the level-2 cache, meaning that, relatively, they do not contribute as much energy to the total processor budget. However, as seen in figures 7 and 10, reconfiguring the level-2 cache can bring significant improvements in energy-delay. Therefore, dynamically reconfiguring the level-2 cache alone results in an overall cache hierarchy energy-delay reduction of 34% compared to a statically configured baseline cache.

VII. Conclusions

This paper has presented a novel configurable cache architecture and a decision tree machine learning model that dynamically predicts the best cache configuration for any application. The main goal is to reduce both dynamic and static energy without losing performance. We have demonstrated that our approach offers reduction in energy-delay product of 17% in the data cache and 34% in the level-2 cache with less than 2% performance degradation in comparison to the baseline cache.

Future work will consider cache reconfiguration on a multicore architecture, where several threads of execution share cache resources. In addition to this, we will investigate resizing the register file, branch target buffers and other processor parameters that are major contributors to power consumption and use compiler knowledge to ease the process of dynamic prediction.

References

[1] S. Manne, A. Klauser, and D. Grunwald, "Pipeline gating: speculation control for energy reduction," *ISCA*, 1998.

[2] J. Abella and A. González, "Heterogeneous way-size cache," in *ICS*, 2006.

[3] D. H. Albonesi, "Selective cache ways: on-demand cache resource allocation," in *MICRO:*, 1999.

[4] L. Chen, X. Zou, J. Lei, and Z. Liu, "Dynamically reconfigurable cache for low-power embedded system," in *ICNC*, 2007.

[5] A. Gordon-Ross, J. Lau, and B. Calder, "Phase-based cache reconfiguration for a highly-configurable two-level cache hierarchy," in *GLSVLSI*, 2008.

[6] A. Gordon-Ross and F. Vahid, "A self-tuning configurable cache," in *DAC*, 2007.

[7] A. Gordon-Ross, F. Vahid, and N. Dutt, "Fast configurable-cache tuning with a unified second-level cache," in *ISLPED*, 2005.

[8] M. Powell, S.-H. Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar, "Gated-vdd: a circuit technique to reduce leakage in deep-submicron cache memories," in *ISLPED*, 2000.

[9] S.-H. Yang, M. D. Powell, B. Falsafi, and T. N. Vijaykumar, "Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay," in *HPCA*, 2002.

[10] C. Zhang, F. Vahid, and R. Lysecky, "A self-tuning cache architecture for embedded systems," *DATE*, 2004.

[11] S.-h. Yang, M. Powell, B. Falsafi, K. Roy, and T. N. Vijaykumar, "Dynamically resizable instruction cache: An energy-efficient and high-performance deep-submicron instruction cache," *Purdue University*, 2000.

[12] C. Zhang, F. Vahid, and W. Najjar, "A highly configurable cache architecture for embedded systems," *ISCA*, 2003.

[13] K. Inoue, T. Ishihara, and K. Murakami, "Way-predicting set-associative cache for high performance and low energy consumption," in *ISLPED*, 1999.

[14] M. D. Powell, A. Agarwal, T. N. Vijaykumar, B. Falsafi, and K. Roy, "Reducing set-associative cache energy via way-prediction and selective direct-mapping," in *MICRO:*, 2001.

[15] C. Zhang, F. Vahid, and W. Najjar, "Energy benefits of a configurable line size cache for embedded systems," in *ISVLSI*, 2003.

[16] K. Flautner, N. S. Kim, S. M. Martin, D. Blaauw, and T. Mudge, "Drowsy caches: Simple techniques for reducing leakage power," *ISCA*, 2002.

[17] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache decay: Exploiting generational behavior to reduce cache leakage power," in *ISCA*, 2001.

[18] Y. Meng, T. Sherwood, and R. Kastner, "On the limits of leakage power reduction in caches," *HPCA*, 2005.

[19] P. J. Joseph, K. Vaswani, and M. J. Thazhuthaveetil, "Construction and use of linear regression models for processor performance analysis," in *HPCA*, 2006.

[20] C. Dubach, T. M. Jones, E. V. Bonilla, and M. F. P. O'Boyle, "A predictive model for dynamic microarchitectural adaptivity control," ser. MICRO, 2010.

[21] C. Dubach, T. Jones, and M. O'Boyle, "Microarchitectural design space exploration using an architecture-centric approach," in *MICRO*, 2007.

[22] E. İpek, B. R. de Supinski, M. Schulz, and S. A. McKee, "An approach to performance prediction for parallel applications," in *Euro-Par*, 2004.

[23] E. İpek, S. A. McKee, R. Caruana, B. R. de Supinski, and M. Schulz, "Efficiently exploring architectural design spaces via predictive modeling," *ASPLOS-XII*, 2008.

[24] P. J. Joseph, K. Vaswani, and M. J. Thazhuthaveetil, "A predictive performance model for superscalar processors," in *MICRO:*, 2006.

[25] B. C. Lee and D. Brooks, "Illustrative design space studies with microarchitectural regression models," in *HPCA*, 2007.

[26] ——, "Efficiency trends and limits from comprehensive microarchitectural adaptivity," in *ASPLOS XIII*, 2008.

[27] B. C. Lee and D. M. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction," in *ASPLOS-XII*, 2006.

[28] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and Jouppi, "Cacti 5.1. Technical Report HPL-2008-20," *HP Laboratories Palo Alto*, 2008.

[29] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Systems Journal*, 1970.

[30] Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, and M. Stan, "Hotleakage: A temperature-aware model of subthreshold and gate leakage for architects," *Technical Report,CS-2003-05*, 2003.

[31] "SPEC Benchmark," http://www.spec.org.

[32] T. Sherwood, S. Sair, and B. Calder, "Phase tracking and prediction," *ISCA*, 2003.

[33] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *SIGKDD Explor. Newsl.*, 2009.

[34] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using simpoint for accurate and efficient simulation," in *SIGMETRICS*, 2003.