Integrating Adaptive On-Chip Storage Structures for Reduced Dynamic Power *

Steve Dropsho†, Alper Buyuktosunoglu‡, Rajeev Balasubramonian†,
David H. Albonesi‡, Sandhya Dwarkadas†, Greg Semeraro‡,
Grigorios Magklis†, and Michael L. Scott†

† Department of Computer Science
‡ Department of Electrical and Computer Engineering
University of Rochester

Abstract

Energy efficiency in microarchitectures has become a necessity. Significant dynamic energy savings can be realized for adaptive storage structures such as caches, issue queues, and register files by disabling unnecessary storage resources. Prior studies have analyzed individual structures and their control. A common theme to these studies is exploration of the configuration space and use of system IPC as feedback to guide reconfiguration. However, when multiple structures adapt in concert, the number of possible configurations increases dramatically, and assigning causal effects to IPC change becomes problematic. To overcome this issue, we introduce designs that are reconfigured solely on local behavior. We introduce a novel cache design that permits direct calculation of efficient configurations. For buffer and queue structures, limited histogramming *permits precise resizing control. When applying these techniques we show energy savings of up to 70% on the individual structures, and savings averaging 30% overall for the portion of energy attributed to these structures with an average of 2.1% performance degradation.*

1 Introduction

The philosophy of high performance microprocessor design has been to push for ever greater performance as the primary goal. Energy consumption used to be a secondary consideration, but with today's smaller and much faster designs energy has become one of the critical system design parameters. There are two basic approaches to reducing energy. The first approach is static: redesign basic hardware with energy efficient features that are always enabled. A complementary approach is to permit dynamic resource allocation and attempt to match the minimal resource requirements of an application. This paper explores the integration of multiple adaptive structures within a high performance general purpose microprocessor.

To achieve high performance across a wide range of applications, key structures in the microarchitecture are generally sized for worst-case applications within technology constraints (area and cycle time). In particular, the caches and the instruction scheduling structures consume a considerable portion of the chip's dynamic energy (20-35% and 55%, respectively, for the Alpha 21264 [21]). Since applications have differing needs, resources are often underutilized and can be reduced in size in order to save energy with minimal impact on performance; though, the particular resources which can be reduced depend on the application. Unfortunately, with multiple adaptive structures, changes to one may affect the behavior of another, either positively or negatively. One approach to minimizing the complexity of controlling multiple adaptive structures is to provide designs in which only local information is required for good configuration decisions, i.e., greedy optimization. Such components can then be used as the basis for simplified meta-control for global optimization.

In this paper, we demonstrate that multiple independent adaptive caches and adaptive scheduling logic can use local information for effective control. We introduce a new adaptive cache structure called an *accounting cache*. Prior adaptive caches use changes in system metrics to initiate a search of the configuration space. Invariably, the system IPC is used as a fail-safe measure of the appropriateness of the cache configuration. In contrast, the accounting cache design leverages LRU state in set associative caches to directly *calculate* ideal configurations for performance and/or energy and eliminate search from the control process. We demonstrate that three instances of the accounting cache, the L1 instruction and data caches and the L2 unified cache, operate simultaneously and independently to save energy with minimal performance impact.

For the instruction scheduling logic, we incorporate a buffer design from [6] and extend a control strategy from [17] to show that *all* buffers involved with scheduling can save energy using this dynamic design. The buffer design is space efficient and electrically isolates partitions of the

*This work was supported in part by NSF grants EIA-9972881, EIA-0080124, CCR–9702466, CCR–9701915, CCR–9811929, CCR-9988361, and CCR–9705594; by DARPA/ITO under AFRL contract F29601-00-K-0182; by an IBM Faculty Partnership Award; and by external research grants from Intel and DEC/Compaq.

buffer so they can be selectively turned off. The control strategy estimates buffer resource requirements through limited histogramming of the occupancy statistics. The intuition is that a full buffer stalls the pipeline, so the buffer should be sized with just enough partitions that the overflow rate is within a specified margin. The histogram information provides the controller with precise information on the minimum size that meets this criterion. The controllers for all structures use a tolerance setting that dictates how aggressively to tradeoff additional delay for lower energy. We evaluate the potential for reduced energy consumption with the adaptive caches and instruction scheduling logic separately and in concert. At the middle tolerance setting, we show energy savings of up to 70% on individual structures with an average savings of 34% when all structures are combined. This energy savings is achieved with an average performance degradation of 2.1%.

Most related work has analyzed one or two individual structures and their control. Albonesi [1] described a cache organization — *selective cache ways* — that provided the ability to reduce energy consumption by restricting the number of ways accessed on a cache access. Balasubramonian et al. [2] expanded this work with a controllable cache and TLB, where both the number of ways and sets of the cache could be set in a limited manner through exploration. Dhodapkar et al. [9] also rely on exploration for reconfiguring the instruction cache but reduces the search overhead with a method to identify and restore previously selected configurations. Powell *et al.* [18] describe the design of an energy-efficient instruction cache whose access mode is dynamically reconfigurable. Buyuktosunoglu *et al.* [6] describe the design and control of an adaptive issue queue which uses IPC as feedback to guide reconfiguration choices. Ghose *et al.* [17] expand this work to improve the control algorithm and apply it to the reorder buffer in addition to the issue queue. We extend the work further to include the physical register files as adaptive structures and incorporate the use of utilization variance rather than averages to control reconfiguration decisions. We discuss the special control needs for this extension.

The rest of the paper is organized as follows. Section 2 describes the overall architecture, highlighting all the adaptive storage structures that we control. Section 3 describes the control algorithm and hardware additions to the design of the adaptive instruction and data caches (the *accounting cache* design). Section 4 describes the design and control of the adaptive buffer structures — the register file, issue queue, and reorder buffer — using *limited histogramming*. Section 5 describes our experimental methodology. We present our evaluation of the potential benefits of the adaptive structures in isolation as well as in concert in Sections 6 to 8. Finally, we compare our design to related work in Section 9 and conclude in Section 10.

2 System overview

Figure 1 shows a schematic of the microarchitecture used in this study. The architecture is representative of a typical out-of-order superscalar design. The adaptable components are the shaded structures. The set of adaptable caches includes the L1 instruction and data caches and the L2 unified cache. All three caches are instantiations of an accounting cache. The set of adaptable storage buffers includes the re-order buffer (ROB), load/store queue (LSQ), integer issue queue (IIQ), floating point issue queue (FIQ), the integer physical register file (IPREG), and the floating point register file (FPREG). The buffers are implemented as RAM structures that are resized by disabling partitions, or groups of entries.

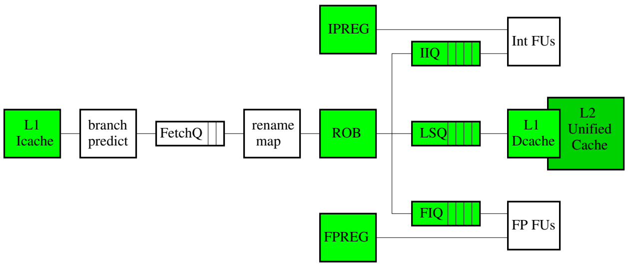

Figure 1. The Base Microarchitecture: Adaptable components are shaded

3 The *accounting cache* design

The *accounting cache* is a reconfigurable cache design with the unique feature that the performance of every possible configuration can be directly calculated from data recorded each interval. This is regardless of the actual configuration of the cache for the interval. Recording the required data is inexpensive and done via a handful of counters. The system designer specifies one configuration as the *base* configuration and the amount of performance degradation that can be tolerated in exchange for lower energy consumption. From these constraints, the accounting cache tracks what the performance of the base cache configuration would have been and reconfigures to the lowest energy configuration that performs within the specified performance degradation limit. This section presents the access protocol to the cache, the additional LRU information required to do the performance tracking, and the performance and energy cost equations.

3.1 Access protocol

The accounting cache design is based on the resizable *Selective Ways Cache* proposed by Albonesi [1]. Resizing is accomplished by disabling ways in a set-associative cache, which reduces energy because fewer ways are activated on a read. Figure 2 shows the data portion of a 4-way set associative cache with ways 2 and 3 disabled (shaded). The tag array can be similarly partitioned (not shown).

The access protocol is shown in Figure 2 and is as follows. The initial access to the cache is the *primary access* or the A access. A hit with this access (*hitA*) returns the data. On a miss with the A access, another access called

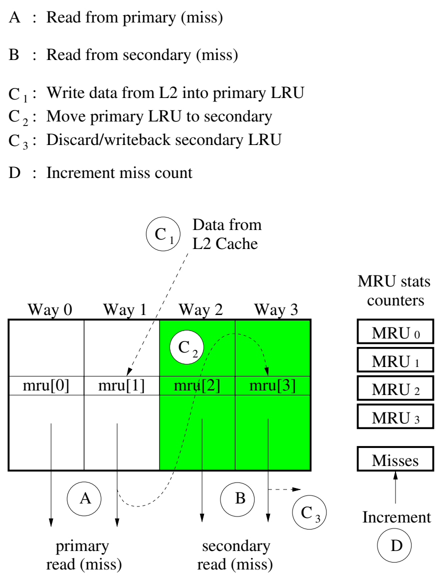

Figure 2. Accounting cache miss on read

the *secondary access*, or B access, is made to the remaining ways. A secondary access stalls subsequent accesses to the primary partition. A hit in the secondary (*hitB*) returns the data but also swaps the block with one from the primary. An access that misses in both the primary and secondary graduates up to the next level in the cache hierarchy. The returned data on a miss is placed in the primary and the displaced block is swapped into the secondary. The displaced block in the secondary is written back if it is dirty. The cache maintains full LRU information for true LRU replacement. We discuss the LRU details in Section 3.2.

The tag array for the cache can be partitioned identically as the cache or the tags can be partitioned independently. For practical considerations, the tags are limited to two configurations: 1) the partitioning is identical to that of the cache (*A-B tags*), or 2) all tag ways are accessed in the primary group (*full tags*). The *full tags* option requires additional tag energy on the primary access (all tag arrays are activated) but cache misses are detected without reading the secondary data cache (which has a high energy cost). Thus, application phases with lots of cache misses will benefit from the *full tags* configuration while phases with few cache misses will prefer the *A-B tag* option. Additionally, the access to the data may be done serially or in parallel to the tags. As one would expect, serially accessing the tag and data will be of most value in the secondary cache but, surprisingly, our results show that the this option sometimes can be of value in the primary cache. Table 1 lists the configuration parameters. As an example, the base configuration of the level 1 data cache in a processor is N-way, *full* tags, and *parallel* tag and data. The three parameters are orthogonal to each other, so a 4-way cache would have $4 \times 2 \times 2 - 1 = 15$ configurations (with a 4-way data primary both *full* and *A-B* tag options are identical).

Table 1. Cache Configuration Parameters

Parameter	Values
Ways (primary)	[1,N]
Tags	{full, A-B}
Tag/Data Access order	{parallel, serial}

3.2 LRU operation and implementation

3.2.1 LRU space requirements

In general, designers refrain from using true LRU replacement due to the additional bits to maintain the LRU state. Simplified algorithms that perform nearly as well as true LRU track only the most recently used way to *not* replace or use a round robin policy. Both of these policies require only $\log_2(N)$ bits where N is the associativity. True LRU, on the other hand, requires $\log(N)$ bits per tag to maintain a full ordering of the sets for a total of $N \log_2(N)$ bits. A 4-way set-associative LRU cache requires 8 bits vs only 2 bits for the other policies, and an 8-way requires 24 versus 3 bits. To put the additional resource requirements into context, in this study we use a 4-way 64KB level-1 data cache with 64 byte lines. In a machine with a 48-bit physical address, the tag for such a cache is 34 bits and the data 512 bits. LRU state adds 2 additional bits per tag or 5.9% to the tag RAM, but only 0.4% additional space when including the data RAM. We accurately account for the energy overhead due to this additional state.

3.2.2 LRU operation

Next-state transitions for true LRU are easily implemented using N counters of $\log_2(N)$ bits each. Assume an 8-way set-associative cache as an example. Starting with the following as the LRU state of the tags for the current cache access, assume a hit in way 4.

Way ID

1	2	3	4	5	6	7	8
LRU State before access							
0	1	2	3	4	5	6	7
LRU State after hit on way 4							
1	2	3	0	4	5	6	7

Upon a hit, all LRU states with a smaller count, i.e., more recently accessed, increment in parallel (ways 1-3). The tag that hit has its LRU state set to zero (way 4). If the LRU state is a higher count then it remains the same. The updated LRU state is written back to the LRU RAM. This update activity occurs for any replacement policy except random, which does not require any state.

3.2.3 Exploiting LRU information

In the following, it simplifies the discussion if we express the LRU order with its dual, the *most-recently-used* (*MRU*) ordering. We denote the most recently used way at a particular cache index as mru_0, the next most recently used as mru_1, and so on. Thus, the least recently used way in an

N-way set associative cache is also mru_{N-1} (the subscript numbering starts at zero).

An LRU replacement policy provides considerable information if we notice that all hits to the most recently used set, mru_0, would be hits in a direct-mapped cache. Similarly, hits to either mru_0 or mru_1 would hit in a 2-way set associative cache. In general, a hit to the n^{th} most recently accessed block mru_n would be a hit in a cache having at least $n + 1$ ways. Counting how many hits occur for each mru state provides sufficient information to reconstruct the hit ratio for *any* partitioning of the cache. The cost is one counter per mru state and one for misses; e.g., a 4-way cache requires a total of only 5 counters to record the activity of the whole cache. For any cache of practical size, the additional energy due to this small set of counters is about 3 orders of magnitude smaller than a cache read access. We account for this (nominal) additional energy in our simulations.

3.3 Performance and energy cost equations

For any configuration, we can directly calculate the total delays in accessing the cache and the energy in the following manner. Assume a given partitioning having c ways in the primary and $N - c$ ways in the secondary. Let us define the number of hits to the primary partition as $hits_A[c]$, to the secondary as $hits_B[c]$, and misses as $Misses$ (we need no configuration parameter here because a miss will miss in all configurations). Assume the operation of the cache is such that every access first tries the A partition, then the B partition on a primary miss, and finally the next level of the memory hierarchy on a true miss. From the MRU counts we can directly calculate how many A and B hits would occur for any configuration given the same access pattern. We can use this information in simple cost functions to directly calculate the delay or energy for the set of accesses. The cost functions are given in Table 2.

Let us denote the primary, secondary, and miss RAM data access delays as d_A^d, d_B^d, and d_M^d, and the corresponding tag access latencies as t_A^d and t_B^d (there is no corresponding tag access delay for a miss), using the superscript d as a mnemonic for *delay*. We also include a bus delay variable bus^d to account for non-RAM access overhead in transfers (bus transfer delay). The energies are likewise labeled d_A^e, d_B^e, d_M^e, t_A^e, and t_B^e. The parameters t_{full}^d and t_{full}^e are the delay and energy of the tags in the *full* configuration. Note that each individual cache will have its own values for these factors. In Table 2, Equations 5a-f detail the costs for delay and energy for a cache configured using *parallel* tag/data and *A-B* tags. Using *A-B* tags means that a second access to the secondary partition must take place on a miss in the primary. In contrast, using *full* tags on the primary access eliminates the access to the secondary on a cache miss. In Equation 5a, the delay on a hit in A is the bus transfer costs bus^d plus the access time to the data RAM with the (faster) tag access time hidden behind the data access. The energy is that of the data and tag RAMs configured with c ways. On a hit in the secondary, the delay includes the access to A and then B. Since blocks must be swapped on a hit to the secondary, the energy includes these writes to the data and tag RAMs. Each write is to only one way and has energy $d_{swap}^e = d_1^e$ for the data RAM and $t_{swap}^e = t_1^e$ for the tag RAM. The remaining equations in Table 2 should be self explanatory. Not shown are the equations for the configuration combinations *parallel/full* and *serial/A-B* as they are easily derived from the above examples.

Table 2. Accounting cache cost functions

General cost functions		
1. $Total_{C_A}[c] = hits_A[c] \times C_A[c]$		
2. $Total_{C_B}[c] = hits_B[c] \times C_B[c]$		
3. $Total_{C_M} = Misses \times C_M$		
4. $Total_C[c] = Total_{C_A}[c] + Total_{C_B}[c] + Total_{C_M}$		
Delay cost of *parallel* tag/data with *A-B* tags		
5a. Delay	A hit	$D_A[c] = bus^d + d_A^d$
5b. Energy		$E_A[c] = d_A^e + t_A^e$
5c. Delay	B hit	$D_B[c] = bus^d + d_A^d + d_B^d$
5d. Energy		$E_B[c] = (d_A^e + t_A^e) + (d_B^e + t_B^e) + (d_{swap}^e + t_{swap}^e)$
5e. Delay	miss	$D_M[c] = bus^d + d_A^d + d_B^d + d_M^d$
5f. Energy		$E_M[c] = E_B[c]$ (see 5d)
Delay cost of *serial* tag/data with *full* tags		
6a. Delay	A or B hit	$D_{A/B}[c] = bus^d + t_{full}^d + d_1^d$
6b. Energy		$E_{A/B}[c] = d_1^e + t_{full}^e$
6c. Delay	miss	$D_M[c] = t_{full}^d + d_M^d$
6d. Energy		$E_M[c] = t_{full}^e + (d_{swap}^e + t_{swap}^e)$

3.4 Accounting and configuration selection

Our goal is to improve energy efficiency with minimal degradation in CPI relative to a specified base system. The strategy is to minimize energy with the constraint that the portion of the memory access time attributable to the cache is within a specified percentage of the delay that the same accesses would have in the base system. The base configuration defines the latency to which the dynamic cache must compare. The performance degradation percentage is called the *tolerance setting*. For this study, we use tolerance values of 1.5%, 6.2%, and 25%, which correspond to the fractions 1/64, 1/16, and 1/4, respectively.

Configurations are selected by gathering MRU (or equivalently, LRU) statistics for an interval and then calculating the delay and energy costs. A hardware timer triggers an interrupt to run a small, fast software cache analysis routine. A PAL-code routine like the Alpha 21264 [8] supports does not have to save register state so the overhead is minimal. The analysis routine reads the MRU register values containing how often blocks were accessed with the corresponding MRU state, and calculates the delay and energy values for all possible cache configurations. For example, if our MRU counts are $mru_0 = 0$, $mru_1 = 0$, $mru_2 = 100$, and $mru_3 = 0$, both a 3- and 4-way cache would have the same performance since all the mru_2 accesses would hit, but the controller selects a 3-way configuration because the energy cost of the accesses is lower. The MRU state counters are cleared by the handler before returning. In our simulations we use an interval of 100,000 instructions between reconfigurations. We estimate a highly tuned handler will be on the order of 100 instructions and have a high degree of parallelism (high IPC). The reconfiguration is accomplished by writing a value to a system register. The energy and delay of the handler is the cost of reconfiguration. We do not include the interrupt handler in our simulations.

We call the delay that the base cache configuration would have incurred for the interval d_{base}. We add d_{base} to an accumulating delay counter D_{base}. This counter maintains the total delay cost for all prior accesses for a base cache configuration. A second counter D_{actual} aggregates the calculated delays d_{actual} per interval for the actual cache configurations used. Cache configurations are selected to maintain the relationship relative to a tolerance setting T of $D_{actual} \leq D_{base} \times (1+T)$, while minimizing energy usage. The difference $((1+T) \times D_{base} - D_{actual})$ is a cache's *delay account value* V^d. The account builds savings (credits) that the controller can spend on additional delay in exchange for lower energy. In addition, a similar *energy account* V^e is kept as a fail-safe in the event of pathological behavior (discussed below).

To select the next configuration, we assume the pattern of accesses in the next interval will be identical to that of the prior one. From this assumption, we estimate the permissible delay for the next interval that is within the tolerance setting, $d_{next} \leq d_{base} \times (1+T) + V^d$ (this includes delay credits, or possibly debits, accumulated in prior intervals via V^d). If the quantity $V^d + d_{base} \times T \leq 0$ then $d_{next} \leq d_{base}$ and the controller is forced to select the fastest possible configuration to make up the debit. Otherwise, the configuration with the lowest energy is selected whose estimated delay cost is within $d_{base} \times (1+T) + V^d$.

A key feature of the cache accounts is that they permit the controller to amortize costs or credits across many intervals. Thus, cache selection can be aggressive and performance glitches due to phase shifts in access patterns will be corrected. As we discuss in our results, in the case of over-performance the extra accumulated margin permits a more aggressive energy saving configuration to be selected periodically, even though the per interval delay for the energy saving configuration is above the per interval tolerance. The delay account V^d also helps at phase shifts. The controller does not need to explicitly detect phases. A significant change in the access pattern will manifest itself in a different ordering of configurations based on their total delay costs for the new access pattern. The delay account ensures that the additional delay incurred during a phase shift is eventually paid for in future configurations. The account permits the controller to atone for guessing wrong at that interval. In the event of pathological, rapid phase shifts when the prior interval is a poor estimate of the next, the delay and/or energy accounts for the cache will become negative. A negative account forces the controller to default to the baseline configuration because that configuration guarantees zero difference in performance. Over successive intervals, credits accrued due to the tolerance factor will eventually repay the debit and make the accounts solvent once again. This important feature limits the performance and energy impact of pathological behavior to that of the baseline performance, but also permits reconfigurations again when (and if) that erratic behavior ends.

The accounting cache design requires a set associative organization in which all of the ways are accessible. This condition is sufficient to simulate how larger cache configurations would have performed on any pattern of accesses. With the LRU (MRU) statistics, reconfiguration decisions are independent of the system-wide IPC effects. This is in sharp contrast to exploration-based cache reconfiguration schemes [1, 2, 9] and is the primary advantage of the accounting cache design when combined with other dynamic structures, which can affect the IPC. The accounting cache design as presented here requires a cache to be set-associative. However, we believe other dynamic energy saving schemes (e.g., [2, 18]) can incorporate the basic accounting concept to enhance their efficiency or, at a minimum, to detect and prevent pathological CPI degradations arising from mismatches between the cache configurations and the application (e.g., possibly due to rapid phase changes in the application).

Finally, the delay calculations are only estimates of the effects on the processor's CPI. There are multiple reasons why: memory accesses account for only a percentage of an application's execution time; the memory hierarchy may have slack relative to the other parts of the processor so an increase in delay may not significantly increase CPI; and parallelism between the caches, particularly between the level-1 instruction and data caches, decouples the effects of delays as well. Selecting cost parameters that assume maximum parallelism ensures that the calculated CPI degradation is an upper bound on the true CPI degradation due to cache reconfigurations. A future direction of study is on how to tighten this bound on the CPI effects by accounting for the above situations more accurately.

4 The adaptive buffer design

Buffers throughout the processor store instructions in order to decouple timing dependencies between the stages, as well as to increase the effective window of available instructions in order to exploit instruction-level parallelism. Our microarchitecture (Figure 1) has individual queues for the separate types of functional units. Specifically, we modified Simplescalar to split its centralized RUU into a set of buffers: a reorder buffer (ROB), one integer issue queue (IIQ), one floating point issue queue (FIQ), and one load/store queue (LSQ). In addition, there are the separate physical integer register file (IPREG) and the floating point physical register file (FPREG). In this study, we assume all buffers are implemented as RAMs, with associative addressing capabilities in the IIQ, FIQ, and LSQ.

To save energy, each buffer's RAM is partitioned by bit-line segmentation. Bit-line segmentation electrically isolates regions of the RAM to reduce dynamic energy on accesses [6]. Figure 3 shows how bit-line segmentation reduces the access energy by reducing the capacitance on the bit-lines. Only the enabled partitions expend dynamic energy [6]. An alternative RAM design based on banking is

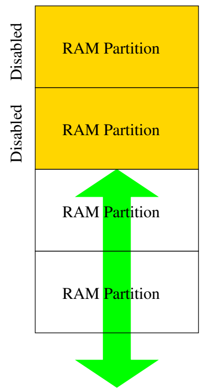

Figure 3. Resizable buffer with 2 partitions diabled

used in [17]. The underlying structure of the RAM does not impact the control. In both designs, the electrically optimal partitioning was found to be the same: 16 entry partitions for the ROB and 8 entry partitions for all other buffers.

4.1 IIQ, FIQ, LSQ, and ROB resizing

When downsizing a buffer we always turn off the partitions in the order of highest address to lowest, and resize up in the reverse order. Restricting the resizing in this manner simplifies the circuitry [6]. However, before downsizing the IIQ or FIQ, we must wait until existing instructions in the partitions to be turned off have issued. Furthermore, we must restrict instructions from being dispatched into these partitions. Additional care must be taken in resizing the ROB and LSQ because of their circular FIFO-like structure. The physical implementation uses a head and tail pointer with wrap-around at the buffer limit. Before resizing, we must ensure the head and tail pointers do not straddle the partition to be disabled otherwise buffered instructions could be caught in the disabled partition [17].

4.2 Register rename operation

Register renaming [16] performs logical (architectural) to physical register mappings, thereby eliminating write-after-read and write-after-write dependences when there are sufficient physical registers. In a processor such as IBM's Power4, which can support up to 200 instructions in flight through the pipeline simultaneously [20], register renaming is critical for exploiting the processor's superscalar capabilities. The approach to register renaming proposed by [16] utilizes the pool of registers more efficiently than alternative designs that maintain a separate pool of architected registers in addition to a register set for renaming. In this design, upon instruction commit the architected register's contents are updated. A design with a single common pool of registers offers better opportunity to turn off more physical registers. However, since the physical registers remain mapped until the logical register is over-written, before buffer resizing can occur, all active registers in the partition to be disabled must be freed by moving their contents to different *physical* registers that will remain active (the *logical* registers are the same).

The rename logic complicates resizing in comparison to the issue queues. In the integer register file, for example, when a load is fetched, the rename logic selects a register from the free list and records the logical to physical mapping in the map table. A mapped physical register becomes free when the following are satisfied:

1. The instruction writing the value has been committed.
2. All in-flight instructions using the value have read it.
3. The physical register has been unmapped by a subsequent write to the same logical register.

Until all three conditions are satisfied, a physical register cannot be freed. It may be possible that a logical register is used early in the program and never assigned again by the compiler. The physical register mapped to that logical register can never be freed. The implication to dynamically resizing the register file is that we cannot guarantee a partition of the register file RAM will ever have all its physical registers unmapped. Because of this issue, simply disabling partitions as is done in the issue queues will not work. This condition would seem to greatly diminish the likelihood of ever being able to downsize the register files.

To turn off a partition in the register file requires the following: remove the registers to be disabled from the free list; move the contents of active registers to be disabled to other registers that will remain enabled, and remove the newly freed registers to be disabled from the free list.

This additional complexity for resizing the register files necessitates a software handler. To move a logical register to a new physical register, the software handler issues a move instruction from the logical register back to itself: *mov r7, r7*. The normal operation of the rename logic will move the contents into a new physical register from the free list and unmap the physical register in the partition to be disabled. The map table contains the information for logical to physical register mappings so the software handler could have direct access to this information. Turning a partition back on is then just a matter of adding the partition's registers back onto the free list. Because the RAM is disabled from the top (highest numbered register) down (lowest), an ID can be set and any register with an address greater than the ID is considered removed from the free list and cannot be selected by the renaming logic.

4.3 Reconfiguration control

The controller is an extension of the design in [17]. In that work, the buffer is sampled at periodic intervals and the number of entries are accumulated in a counter. At the end of the interval, a simple shift of the count provides the average buffer occupancy. The buffer is sized to the number of partitions that will hold the average occupancy. An *overflow counter* ensures against a buffer being sized too small. Every cycle the buffer is full increments the overflow counter. When the count reaches a threshold it triggers an immediate upsizing of one partition.

Our implementation uses a different emphasis for the design. The intuition is that we want to size the buffer such that overflows occur with a frequency below a set threshold during the interval. In a finely partitioned buffer, choosing the average occupancy size means that half of the time we would then actually require a size greater than the average. We take the position that the upper tail of the occupancy distribution is the proper metric. The proper sizing is that which selects a sufficient number of partitions such that the portion of the distribution that would extend beyond the resized buffer (i.e., the overflows) is less than the threshold. We histogram the occupancy at the granularity of the partitions, p entries per partition. In particular, for a buffer with four partitions each with p elements, there are three counters. The first counter h_0 increments when the occupancy N is greater than or equal to p, the next counter h_1 increments when $N \geq 2p$, the third counter increments when $N \geq 3p$. The counters are associated logically with the buffer and track the true number of entries occupied and not the actual locations occupied in each partition (i.e., the histogram is a virtual compaction of the queue). Since we are only concerned if the overflow threshold would have been exceeded, the counters saturate at the overflow limit. The saturation reduces the number of cycles the counter is active. We downsize the buffer to the partition whose counter has not saturated. The highest numbered histogram counter of the active partitions simultaneously acts as the overflow counter. We also immediately upsize when the overflow threshold is reached. We use the same three tolerance settings as in the cache simulations: 1.5% (1/64), 6.2% (1/16), and 25% (1/4). The overflow threshold is the tolerance fraction of the interval period, e.g., 6.2% of 8K cycles.

5 Evaluation methodology

Our evaluation methodology uses Simplescalar [5] for the Alpha AXP instruction set and the Wattch power model [4]. We simulate an out-of-order superscalar processor similar to the Alpha 21264 in that there are separate issue queues for the different types of functional units. Table 3 lists the microarchitectural parameters and Table 4 lists the benchmark suite.

We model the memory hierarchy in detail to account for all actions and side-effects of the configurable cache operation, including the swapping of cache blocks between primary and secondary partitions. The TLBs are not configurable in this study. Our timing analysis reveals that serial accesses have a latency that is approximately 1.6 that of a parallel access. We use this multiplier to set the access time for a cache configured to serially access tag and data, thus, the access latency of the L1 caches increases to 4 cycles from 2 in serial tag/data mode and the L2 access latency increases to 19 from 12 cycles.

Table 3. Architectural parameters

Fetch queue	8 entries
Branch predictor	comb. of bimodal and 2-level gshare; bimodal/Gshare Level 1/2 entries- 2048, 1024 (hist. 10), 4096 (global); Combining pred. entries - 1024; RAS entries - 32; BTB - 4096 $\times$ 2-way
Branch mispred. latency	10 cycles
Fetch, decode, width	4 instructions
Reorder buffer	128 entries
Integer issue	32 entries
Floating point issue	32 entries
Physical INT regs	96
Physical FP regs	96
Load entries	32 entries
Store entries	32 entries
Instruction TLB	256 (64 $\times$ 4-way) 8K pages, 30 cycle miss
Data TLB	512 (128 $\times$ 4-way) 8K pages, 30 cycle miss
Memory latency	80 cycles
L1 I-cache	64 KB, 4-way, 64B line, 2 cycle
L1 D-cache	64 KB, 4-way, 64B line, 2 cycle
L2 unified	2 MB 8-way, 128B line, 12 cycle

6 Accounting cache results

Due to space constraints, we condense the results for the benchmark suite into Table 5 using arithmetic averages. We summarize the energy and delay data in Figure 4. In Table 5 the first line lists the thresholds used. The thresholds are 1.5%, 6.2%, and 25%, which approximately correspond to 1/64, 1/16, and 1/4 (so the controller can use shift as a fast divide). The results compare the adaptable cache to a baseline configuration using the maximum number of ways with parallel tag/data access. We also compare against a base L2 with serial tag/data access.

6.1 Instruction cache

The instruction cache data is the first group of data in Table 5. The energy saving is 54.3% at 1.5% and 58.6% at tolerances of 6.2% and 25%. The reason for the large savings is clear from the breakdown of the average time spent in each of the cache configuration parameters. The most revealing metric is that the average number of ways is no larger than 1.2 (out of 4) across the tolerance levels. Thus, the minimum configuration of a direct-mapped 16 KB instruction cache is generally sufficient for the benchmarks and the controller correctly configures the cache to use minimum energy. An exception is *vortex* which uses 2.6 ways on average at 1.5% tolerance saving only 25% energy. At the lowest tolerance setting, the *full* tags configuration is selected during phases of cache misses to eliminate the delay for unnecessary accesses to the secondary and keep within tolerance.

We selected a relatively large 64 KB cache that is similar in size to the Alpha 21264 [15]. An drastically smaller cache would have been needed for the benchmarks to stress it. We feel that the data cache results showcase the controller's abilities appropriately. We report the instruction cache energy contribution so its effects can be judged relative to the total energy. The row labeled *Account Used* is the portion of the extra cycle account actually used by the controller. Even at the highest tolerance of 25% degra-

Table 4. Benchmarks

Benchmark	Suite	Datasets	Instruction Window	64KB 4-way DL1 miss rate
em3d	Olden	20K nodes, 20 iters	1000M-1100M	23%
health	Olden	4 levels, 1000 iters	80M-140M	18%
mst	Olden	2K nodes	500M-600M	2%
compress	SPEC95 INT	ref	1900M-2100M	11%
gcc	SPEC95 INT	ref	1650M-1750M	6%
parser	SPEC2K INT	ref	2000M-2200M	3%
perlbmk	SPEC2K INT	ref	2000M-2200M	1%
twolf	SPEC2K INT	ref	1000M-1200M	5%
vortex	SPEC2K INT	ref	2000M-2200M	1%
vpr	SPEC2K INT	ref	2000M-2200M	2%
applu	SPEC95 FP	ref	200M-400M	3%
art	SPEC2K FP	ref	300M-500M	22%
swim	SPEC2K FP	ref	1000M-1200M	8%
wave5	SPEC95 FP	ref	200M-400M	1%

dation, the controller only needed to withdraw less than 1/25*th* of the accounts value.

6.2 Data cache

The level-1 data cache has more interesting behavior. As the tolerance level is increased, the energy savings also increase from 29.6% up to 45.2% due to adjustments in all three configuration parameters. The *A-B* tag option is selected over 30% of the time at the 6.2% and 25% tolerance settings. Somewhat surprisingly, the serial tag/data option is sometimes selected (2.9%) at the aggressive tolerance setting. The most energy savings, however, come from reducing the ways from 4 ways to nearly a direct-mapped cache, 1.3 ways on average at the 6.2% tolerance level. Notice that there are very few hits in the secondary (*B*) partition. The controller will always select configurations to keep these accesses to a minimum.

6.3 L2 unified cache (parallel tag/data)

The first set of data for the L2 cache is relative to a high performance base configuration using parallel tag and data access. We can see that the controller aggressively uses the *A-B* and *serial* configuration options to save energy and is less aggressive at decreasing the number of ways (to 6.1 out of 8). The reason is that serializing the tag and data accesses is, in general, the single most effective means of decreasing energy consumption. The large memory access latency of 80 cycles provides a significant amount of credit to the L2 cache account that it can then trade for serializing the tags and data. The L2 controller uses over half of its tolerance limit (16% out of 25%), but this results in only a 3.9% average slowdown. As mentioned previously, a cycle slowdown in the memory hierarchy usually does not translate to a similar slowdown in the pipeline.

6.4 L2 unified cache (serial tag/data)

In Table 5, configuring for serial tag/data results in a small net loss of energy relative to the serial tag/data base configuration. The reason is due to the large difference between the tag energy and that of a data block access. In this architecture, reading all the tags requires only about 1/4th the energy of reading one data block. Recall that on a miss blocks are swapped between the primary and secondary partitions. With such a disparity between energies, any additional data cache activity swamps any energy savings from partitioning the tags. The controller correctly detects this state of affairs and defaults to the base configuration. However, in this configuration the extra LRU bits of the accounting cache require additional energy not in the baseline cache and this results in a small net loss of 1.1% across the benchmarks.

A conclusion one might draw is that parallel tag/data access is not an attractive option. This conclusion is not necessarily correct. Table 6 lists the CPI values for each of the benchmarks for both parallel tag/data access and serial tag/data access for our base system *without* adaptable caches. The ratio of the serial vs parallel base performance is shown in the last row. Most applications show little performance impact. However, *compress* and *health* show 11% and 30%, respectively. Thus, some applications can run significantly faster if the L2 cache can be optionally configured for parallel tag/data as well as serial tag/data. If the additional performance is important then the dynamic cache can realize energy savings of 49.1% in the L2 for *compress* at the small 1.5% delay tolerance setting (and only 0.3% actual slowdown). For fair comparison, the serial tag/data option offers an 85% energy savings (about twice) but incurs the 11% slowdown.

Figure 4 graphs the relative energy savings for the different cache levels and also the aggregate relative performance degradation. The first three groups of bars are for the individual cache levels. The fourth set is the relative energy savings for the complete cache hierarchy assuming an L2 with parallel tag/data access. In the third grouping, the lighter portionis the relative energy savings excluding the instruction cache (thus, the savings are relative to the data and L2 base energy). The dark portion is the additional savings if the instruction cache is included. The rightmost set of bars is the system performance degradation for each of the tolerance settings that was reported in Table 5. Selecting the modest tolerance setting of 6.2% results in over 40% energy savings relative to the caches, but incurs less than a maximum of 2.8% performance degradation across all the benchmarks with the average at 1.1%.

The average performance degradations are well below the tolerance setting. While the conservative design of the controller is such that this relationship always holds, feed-

Table 6. Base Configuration CPI: Parallel vs Serial L2 Tag/Data

	em3d	*health*	*mst*	*compress*	*gcc*	*parser*	*perlbmk*
Parallel	1.2270	1.7912	0.3170	0.6074	0.6173	0.6328	0.6247
Serial	1.2720	2.3252	0.3174	0.6742	0.6182	0.6731	0.6372
S/P Ratio	1.0367	1.2981	1.0013	1.1100	1.0015	1.0637	1.0200

	twolf	*vortex*	*vpr*	*applu*	*art*	*swim*	*wave5*
Parallel	0.6513	0.4319	0.7001	0.5263	0.8426	0.6311	0.3488
Serial	0.7108	0.4646	0.7143	0.5266	0.8756	0.6342	0.3495
S/P Ratio	1.0914	1.0757	1.0203	1.0006	1.0392	1.0049	1.0020

Table 5. Energy, delay, and percent of time a configuration option is selected, averaged across benchmarks

Averages Across Benchmarks				
Threshold		1.5 %	6.2 %	25.0 %
Delay	Increase	0.3 %	1.1 %	3.9 %
Instruction L1 Cache				
Energy	Savings	54.3 %	58.6 %	58.6 %
Tags	full	5.4 %	0.1 %	0.1 %
	A-B	94.6 %	99.9 %	99.9 %
Data	parallel	100.0 %	100.0 %	100.0 %
	serial	0.0 %	0.0 %	0.0 %
Ways	Ave	1.2	1.0	1.0
Hits	B	0.2 %	0.4 %	0.4 %
Account	Used	0.5 %	0.9 %	0.9 %
Data L1 Cache				
Energy	Savings	29.6 %	42.1 %	45.2 %
Tags	full	89.7 %	68.3 %	61.8 %
	A-B	10.3 %	31.7 %	38.2 %
Data	parallel	99.9 %	99.7 %	97.1 %
	serial	0.1 %	0.3 %	2.9 %
Ways	Ave	2.0	1.3	1.2
Hits	B	0.7 %	2.0 %	2.2 %
Account	Used	1.3 %	4.4 %	6.6 %
Unified L2 Cache (Parallel Tag/Data Base)				
Energy	Savings	25.5 %	41.1 %	63.0 %
Tags	full	74.9 %	67.9 %	60.6 %
	A-B	25.1 %	32.1 %	39.4 %
Data	parallel	94.7 %	66.9 %	32.9 %
	serial	5.3 %	33.1 %	67.1 %
Ways	Ave	5.5	6.0	6.1
Hits	B	1.9 %	1.8 %	1.0 %
Account	Used	1.5 %	6.0 %	16.0 %
Unified L2 Cache (Serial Tag/Data Base)				
Energy	Savings	-1.1 %	-1.1 %	-1.1 %
Tags	full	67.5 %	67.8 %	67.8 %
	A-B	32.5 %	32.2 %	32.2 %
Data	parallel	0.0 %	0.0 %	0.0 %
	serial	100.0 %	100.0 %	100.0 %
Ways	Ave	6.7	6.7	6.7
Hits	B	0.3 %	0.3 %	0.3 %
Account	Used	0.5 %	0.5 %	0.5 %

back to tighten this bound would in improve the energy savings. We are exploring how to measure the actual delay costs as they relate to the instruction commit rate. Information from *critical loads* [10, 19] may help.

Due to the mismatch between reward and penalty in a serial tag/data access cache, the adaptive accounting cache design is most appropriate when a cache offers the parallel tag/data access option. As an extended policy, a meta-controller could activate the full LRU state and the adaptable capabilities of an L2 cache if high performance is required, but revert back to the simpler replacement policy (to save the 2% energy) when serial tag/data is determined to be sufficient. An interesting feature of the accounting cache is that its controller can easily determine the performance and energy trade offs and notify the meta-controller if, for instance, certain programmed pre-conditions are met.

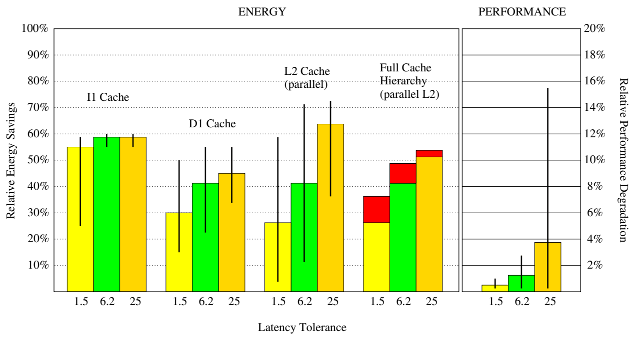

Figure 4. Relative cache energy usage averaged across all benchmarks

7 Dynamic buffer results

Figure 5 shows the relative sizings for each of the buffers averaged across the integer benchmarks in the top graph and across the floating point benchmarks in the bottom graph. Each group of bars represents one of the buffers and the three bars in the group are for each of the tolerance settings, 1.5%, 6.2%, and 25% from left to right. The difference in sizings is minimal between tolerances 1.5% and 6.2%. The differences are small in system performance as well (far right) showing degradations of 0.3% and 0.8%. When the tolerance setting is pushed to 25% the structures are shrunk more aggressively and the average delay increases to 11.9%.

In the floating point benchmarks, again there is nominal difference between the two lowest tolerances of 1.5% and 6.2%, and the highest threshold trades significant performance (11.1%) to shrink the structures. Overall, these results track those in [17] which looked at a combined integer and floating point issue queue, the LSQ, and the ROB. In that study, the authors used a similar range of tolerances and the results showed similar variance in the performance degradation across the applications.

Both *mst* and *wave5* have large performance degradations at the 25% tolerance level, 41% and 39%, respectively. In exploring this behavior, we discovered that in

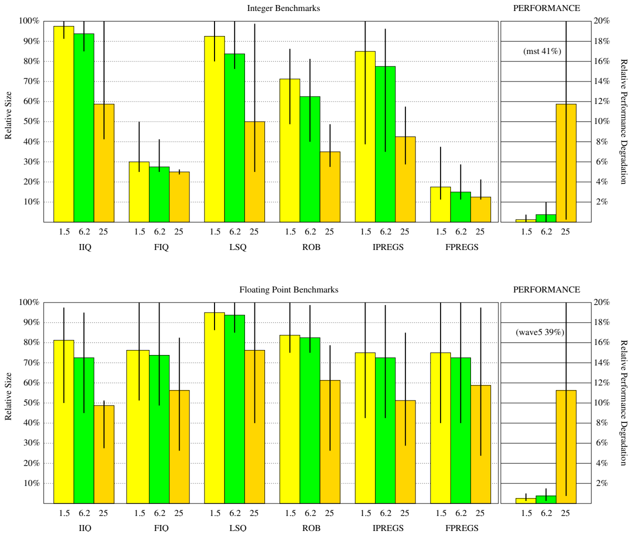

Figure 5. Buffer Relative Sizing and Performance

both benchmarks about half of the degradation was due to adapting the register files and the other half was due to adapting the queues. The performance degradation effects of adapting both sets of structures simultaneously are additive for these applications. This coupling of the effects appears to be fairly infrequent but it can occur and we specifically included *mst* and *wave5* to highlight this behavior. The adaptive cache takes into consideration some of the inter-cache effects via selecting proper values in the cost functions. Unfortunately, there is no equivalent inter-buffer feedback. For the buffers, the tolerance setting effectively bounds the per buffer effects on performance, but the possibility exists for the delay effects to be additive. Thus, the tolerance must be set conservatively. For future work, a method of assigning accurate performance costs to each of the buffers is needed so a system wide tolerance setting can be used.

8 Integrated system results

Figure 6 shows the results of combining all the adaptable structures in the system. The energy savings are shown on the left. In all groups, the energy savings are relative to the base energy of the components in the group. The first two groups of bars are the aggregated results for the cache hierarchy and buffers, respectively. The overall savings for the caches and buffers are 26%, 34%, and 48% for the tolerances 1.5%, 6.2%, and 25%, respectively. The performance degradation has high variability at the aggressive tolerance setting of 25%. The raw data of the average number of cache ways and percent of buffer size activated is shown in Table 7 for all applications. Results for the 1.5% tolerance setting are omitted to save space. The large performance degradations at the 25% tolerance level are due to the additive delays between the reconfigurable buffers.

To integrate the caches with the buffers we had to eliminate the need for tracking system IPC to guide cache configuration decisions as is done universally in prior work. By

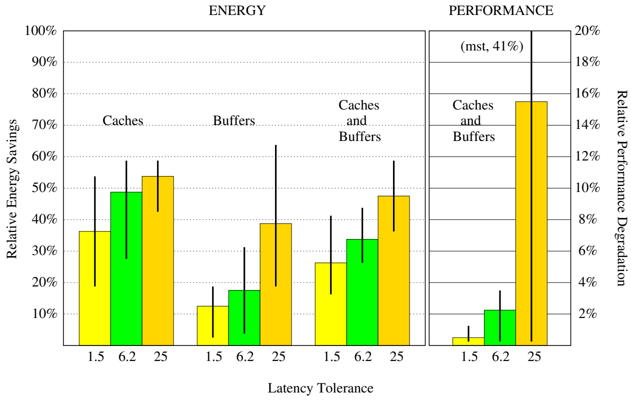

Figure 6. Summary of results

adding the accounting cache, each dynamic structure relies solely on local information for its resizing control. The disadvantage of this approach is that performance degradations due to reconfiguration actions can be additive (e.g., *wave5*), but the configuration controllers cannot account for this possibility. An important result of this exercise to integrate so many dynamic structures is that it highlights the fact that some method of global coordination is necessary. A global controller will require mechanisms in microarchitecture to quantify these coupled effects between the reconfigurable structures. The ultimate goal should be to control the *variability* of the performance degradation when reconfiguring to save energy. Ideally, given a maximum performance degradation target the system should find energy savings without violating this constraint. Current reconfiguration techniques (ours included) cannot offer such a guarantee. We feel this is an important direction for future research.

9 Related work

This paper integrates a large number of dynamic structures that adapt independently to save energy with modest performance impact. The related work can be split into two groups: dynamic energy efficient caches and dynamic instruction scheduling logic. Distinct from dynamic control are static methods to improve energy efficiency. The static methods, such as subbanking, bit-line segmentation, and Gray coding, are orthogonal to the dynamic methods described here and both can be used together.

In addition to the related work [1, 2] described in Section 3, Dhodapkar and Smith [9] extend the work in [2]. As in [2], the configuration space is searched and a configuration is selected based on observing the system IPC. The extension is a method to generate a unique signature that they associate with the configuration parameters and store in a table. If the same working set signature occurs then the stored configuration can be read and directly applied without another search. Due to space limitations, we do not compare the accounting cache to the above designs. However, results not reported here have shown the energy savings of the accounting cache to be comparable to those of [2] for similar cache architectures. We consider the pri-

Table 7. Summary results' raw resizing data

App	Perf	Ave cache ways			Buffer size (%)					
	(%)	IL1	DL1	L2	IIQ	IPREG	FIQ	FPREG	ROB	LSQ
Tolerance setting of 6.2%										
em3d	2.2	1.0	1.0	7.0	89	82	50	50	75	100
health	3.0	1.0	1.6	7.3	99	51	25	14	39	76
mst	2.7	1.0	1.0	8.0	97	99	25	12	75	87
compress	1.6	1.0	1.2	4.5	86	95	35	16	76	79
gcc	0.2	1.0	1.0	2.4	98	35	25	12	50	100
parser	3.6	1.0	1.1	6.0	92	85	25	12	64	80
perlbmk	2.0	1.0	1.3	2.9	86	63	25	12	44	68
twolf	2.6	1.0	1.7	6.4	97	90	26	15	64	79
vortex	2.7	1.0	1.2	5.1	95	85	25	12	67	93
vpr	2.1	1.0	2.1	6.5	92	96	42	30	82	97
applu	1.6	1.0	1.5	8.0	71	55	96	98	82	99
art	1.8	1.0	1.0	7.3	78	88	50	40	76	90
swim	2.0	1.0	1.0	8.0	46	41	82	99	79	86
wave5	0.8	1.0	1.4	4.7	96	99	96	76	99	99
AVE	2.1	1.0	1.3	6.0	89	70	56	48	72	88
Tolerance setting of 25%										
em3d	4.8	1.0	1.0	8.0	81	76	50	50	75	100
health	15.9	1.0	2.0	7.5	99	48	25	12	37	73
mst	40.9	1.0	1.0	8.0	44	37	25	12	27	25
compress	14.6	1.0	1.3	3.7	43	48	25	12	37	39
gcc	0.3	1.0	1.0	2.1	72	30	25	12	49	99
parser	16.0	1.0	1.0	6.5	60	59	25	12	45	54
perlbmk	13.5	1.0	1.0	2.4	37	28	25	12	22	30
twolf	13.6	1.0	1.6	6.8	71	51	25	12	36	44
vortex	25.7	1.0	1.0	5.9	45	35	25	12	29	47
vpr	10.9	1.0	1.3	8.0	70	60	27	22	48	59
applu	11.3	1.0	1.0	8.0	45	28	81	84	56	74
art	7.4	1.0	1.0	7.4	44	85	44	39	72	89
swim	2.7	1.0	1.0	8.0	42	41	83	98	79	82
wave5	39 4	1.0	1.0	2.4	26	30	31	25	27	38
AVE	15.5	1.0	1.2	6.1	60	47	44	38	50	69

mary contribution of the accounting cache design to be its independence from the system IPC which enables the integration of the dynamic caches with the dynamic buffers.

As mentioned previously, the dynamic RAM design is from Buyuktosunoglu et al. [6]. The controller design [7] shares similar features to the controller of Ponomarev et al. [17], but [17] adds an important upsizing reflex that quickly increases the buffer when metrics indicate it is too small. We extend this work by using histogramming to record the occupancy. We feel the histogram is more robust relative to the average occupancy metric because the histogram reveals the tails of the occupancy distribution. This nuance is most significant when the partitioning is at a fine granularity. Folegnani and Gonzalez [12] study resizing the issue queue and similarly use the system IPC to detect if resizing is needed.

Powell et al. [18] use *selective direct-mapping* to reduce energy on accesses to set associative caches. The method in [18] accesses the tags in full on the primary access, but only reads data from one way. The equivalent configuration in our study is *full* tags and 1-way for the primary partition. The difference in their work is that the low order of the address bits determine which partition acts as the primary partition. Conceptually, this is an extension to our definition of a primary and secondary partitioning. The *selective direct-mapping* cache has tables to record which lines exhibit thrashing behavior and should use the set associativity to mitigate the problem. The authors report that *swim* exhibits pathological behavior that results in significant slowdown on an 8-way cache. Accounting techniques could be added to detect this behavior and reconfigure the cache to avoid this pathological case of thrashing.

Energy due to subthreshold leakage current is expected to become a signficant factor in the near future [3]. Most techniques gate the power to turn off portions of the processor [13, 14, 22]. With this technique data in storage elements is lost. This effect is not an issue when resizing the buffers as done in this study and the unused portion of the buffers can be power gated. For reducing static energy in caches, the *drowsy cache* design proposed by Flautner *et al.* [11] significantly reduces the leakage current while maintaining the stored state. The accounting cache could be built with this circuit technology.

10 Conclusions

Dynamically resizing on-chip storage structures can result in energy savings in the processor. This study integrates the most extensive set of dynamic structures in one system, to date. Our goal was to explore the issues that arise in the simultaneous control and operation of these structures. Our approach uses local information at each component in order to allow independent reconfiguration decisions. This approach explicitly decouples each structure and lets the controllers make greedy control decisions.

We introduced the *accounting cache* design as the dynamic cache component. The accounting cache uses full LRU state to reliably account for energy consumption and delay attributable to each cache. The design uses an *account* to build performance equity to apply toward aggressive energy configurations and as a mechanism to shut off reconfiguration when the access behavior is unpredictable. A *tolerance* metric is used to control the amount of performance degradation (and thereby the size of the account) permissible. The contribution of this design is its ability to directly calculate the effect of different configurations relative to some base configuration and to protect against pathological behaviors.

We also refined and extended prior work in dynamic instruction scheduling buffers to include the physical register files and to take the variance in the utilization of the structures into account (rather than relying on averages). Using a tolerance setting to control the aggressiveness of downsizing the buffers for energy efficiency, the six buffers adapted independently to the changing needs of the applications and with minimal slowdown in most applications. We show how to disable physical registers via injected *MOV* instructions that automatically update the logical to physical register mappings.

When using these designs for all levels of the instruction and data caches, the issue queues, reorder buffer, and register files, we show energy savings of up to 70% on the individual structures, and savings averaging 30% overall for the portion of energy attributed to the adaptive structures. These savings were achieved with an average performance degradation of 2.1% (and a maximum performance degradation of 3.6%) when using a 6.25% tolerance metric for our benchmark suite. While our results show that performance degradation is controllable (via the tolerance settings) and can be minimized, they also reveal the sensitivity of the integrated system to the tolerance setting and application behavior. Future work will explore how the current mechanisms perform under multi-tasking/multithreaded workloads and we are experimenting with methods to ensure performance degradation target can be met deterministically in exchange for energy savings.

References

[1] David H. Albonesi. Selective cache ways: On-demand cache resource allocation. In *32nd International Symposium on Microarchitecture*, November 1999.

[2] Rajeev Balasubramonian, David H. Albonesi, Alper Buyuktosunoglu, and Sandhya Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *33rd International Symposium on Microarchitecture*, 2000.

[3] S. Borkar. Design challenges of technology scaling. In *IEEE Micro*, July 1999.

[4] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *27th Annual International Symposium on Computer Architecture*, June 2000.

[5] Doug Burger and Todd Austin. The simplescalar toolset, version 2.0. Technical Report TR-97-1342, University of Wisconsin-Madison, June 1997.

[6] Alper Buyuktosunoglu, David H. Albonesi, Stanley Schuster, David Brooks, Pradip Bose, and Peter Cook. A circuit level implementation of an adaptive issue queue for power-aware microprocessors. In *11th Great Lakes Symposium on VLSI*, March 2001.

[7] Alper Buyuktosunoglu, Stanley Schuster, David Brooks, Pradip Bose, Peter Cook, and David H. Albonesi. An adaptive issue queue for reduced power at high performance. In *Workshop on Power-Aware Computer Systems, in conjunction with the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[8] Compaq. Alpha 21264 Microprocessor Hardware Reference Manual. Technical report, Compaq Computer Corporation, July 1999.

[9] Ashutosh S. Dhodapkar and James E. Smith. Managing multi-configurable hardware via dynamic working set analysis. In *29th Annual International Symposium on Computer Architecture*, 2002.

[10] B. Fisk and I. Bahar. The non-critical buffer: Using load latency tolerance to improve data cache efficiency. In *IEEE International Conference on Computer Design*, October 1999.

[11] Kristian Flautner, Nam Sung Kim, Steve Martin, David Blaauw, and Trevor Mudge. Drowsy caches: Simple techniques for reducing leakage power. In *29th Annual International Symposium on Computer Architecture*, May 2002.

[12] Daniele Folegnani and Antonio Gonzalez. Energy effective issue logic. In *28th International Symposium on Computer Architecture*, 2001.

[13] Heather Hanson, M. S. Hrishikesh, Vikas Agarwal, Stephen W. Keckler, and Doug Burger. Static energy reduction techniques for microprocessor caches. In *2001 International Conference on Computer Design*, September 2001.

[14] Stefanos Kaxiras, Xhigang Hu, and Margaret Martonosi. Cache decay: Exploiting generational behavior to reduce cache leakage power. In *International Symposium on Computer Architecture*, 2001.

[15] R. E. Kessler, E. J. McLellan, and D. A. Webb. The Alpha 21264 Microprocessor Architecture. In *1998 International Conference on Computer Design*, October 1998.

[16] Mayan Moudgill and Keshav Pingali and Stamatis Vassiliadis. Register renaming and dynamic speculation: an alternative approach. In *36th Annual International Symposium on Microarchitecture*, March 1993.

[17] Dmitry Ponomarev, Gurham Kucuk, and Kanad Ghose. Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. In *34th International Symposium on Microarchitecture*, December 2001.

[18] Michael Powell, Amit Agrawal, T. N. Vijaykumar, Babak Falsafi, and Kaushik Roy. Reducing set-associative cache energy via selective direct-mapping and way prediction. In *34th Annual International Symposium on Microarchitecture*, December 2001.

[19] S. T. Srinivasan and A. R. Lebeck. Load latency tolerance in dynamically scheduled processors. In *Journal of Instruction-Level Parallelism*, October 1999.

[20] Joel M. Tendler, Steve Dodson, Steve Fields, Hung Le, and Balaram Sinharoy. POWER4 System Microarchitecture. Technical report, IBM Server Group, October 2001.

[21] K. Wilcox and S. Manne. Alpha processors: A history of power issues and a look to the future. In *Cool-Chips Tutorial*, November 1999.

[22] Se-Hyun Yang, Michael D. Powell, Babak Falsafi, Kaushik Roy, and T. N. Vijaykumar. An integrated circuit/architecture approach to reducing leakage in deep-submicron high performance I-caches. In *Seventh International Symposium on High-Performance Computer Architecture*, January 2001.