RC22856 (W0307-167) July 24, 2003
Computer Science

IBM Research Report

Adaptive Processing: Dynamically Tuning Processor Resources for Energy Efficiency

David H. Albonesi, Rajeev Balasubramonian, Steven G. Dropsho, Sandhya Dwarkadas, Eby G. Friedman, Michael C. Huang, Volkan Kursun, Grigorios Magklis, Michael L. Scott, Greg Semeraro
Departments of Electrical and Computer Engineering and of Computer Science
University of Rochester

Pradip Bose, Alper Buyuktosunoglu, Peter W. Cook, Stanley E. Schuster
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Adaptive Processing: Dynamically Tuning Processor Resources for Energy Efficiency

David H. Albonesi, Rajeev Balasubramonian, Steven G. Dropsho, Sandhya Dwarkadas,
Eby G. Friedman, Michael C. Huang, Volkan Kursun, Grigorios Magklis,
Michael L. Scott, and Greg Semeraro
Departments of Electrical and Computer Engineering and of Computer Science
University of Rochester

Pradip Bose, Alper Buyuktosunoglu, Peter W. Cook, and Stanley E. Schuster
IBM T.J. Watson Research Center

The productivity of modern society has become inextricably linked to its ability to produce energy-efficient computing technology. Increasingly sophisticated mobile computing systems, powered for hours solely by batteries, continue to rapidly proliferate throughout society, while battery technology improves at a comparably slow pace. In large data centers, handling for example online orders for a .com company or sophisticated web searches, row upon row of tightly packed racks of computers may be warehoused in a city block. Microprocessor energy wastage in such a facility directly translates into higher electricity bills, and even getting sufficient electric supply from utilities to power such a center is no longer a given. Given this situation, energy efficiency has rapidly ascended to the forefront of modern microprocessor design.

Adaptive processing is one approach to improving microprocessor energy efficiency. In this technique, major microprocessor resources are dynamically tuned during execution to better match varying application needs [1, 2]. This is in contrast to the fixed resources supplied at design time in a conventional microprocessor, and to common techniques that turn off idle sections of a processor. By presenting the application with the right amount of hardware at the right time, a significant reduction in energy can potentially be achieved. The challenges in adaptive processing are in achieving this greater efficiency with reasonable hardware and software overhead, and doing so without undue performance loss. Unlike the field of reconfigurable computing, adaptive processing attempts not to stray far from the dynamic superscalar design approach that has been successfully used in many generations of

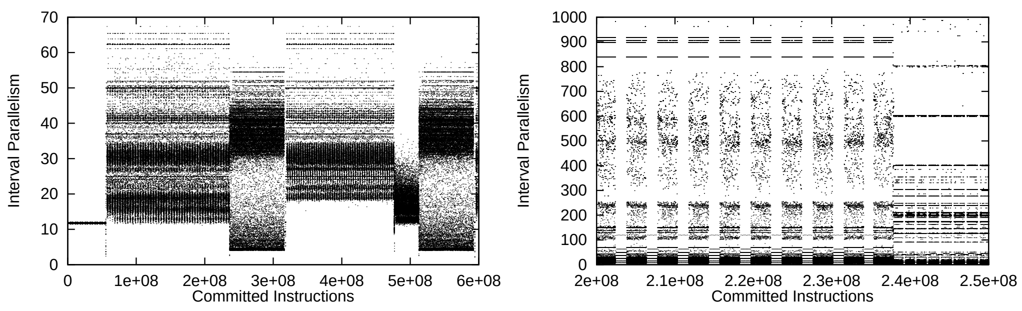

Figure 1. Interval parallelism for SPEC95 benchmark *ijpeg* for an interval of 2000 instructions (left) and 100 instructions (right).

general-purpose processors. Rather, the goal is to maintain this general approach while limiting the extra overhead (area, performance, and energy) of adaptive processing to a few percent. It is this challenge, coupled with the increased emphasis on power-aware microprocessor research in recent years, that has spurred much interest in adaptive processing in the past five years. This article discusses the tradeoffs in adaptive processing and in the process, presents a sampling of the major research findings, with a look towards the growing need to apply such techniques in future technologies.

1 Varying application behavior

Adaptive processing exploits the fact that application needs for particular hardware resources (such as caches, issue queues, and registers) within a dynamic superscalar processor may vary significantly from application-to-application, and even within the different *phases* of a given application. There have been several studies that have demonstrated this dynamic application behavior. Figure 1 shows the behavior of the SPEC95 application *ijpeg* from one of the earliest studies [13]. The graphs show the *interval parallelism* or the parallelism (total committed instructions over total cycles) achieved every set number of instructions. The graph on the left is for an interval of 2000 instructions while 100 instructions is used on the right. The modeled machine has almost infinite resources and perfect caches in order to isolate parallelism limits to those inherent in the application.

In this application, there are multiple levels of phases apparent from these graphs. From the left side of Figure 1, there are major phase shifts that are occurring at the granularity of millions of instructions. The right side of this figure shows that within these major phases are finer-grain ones that last on the order of 10's of thousands of instructions. Within these phases, parallelism varies widely, and this variation is shown in [13] to have a

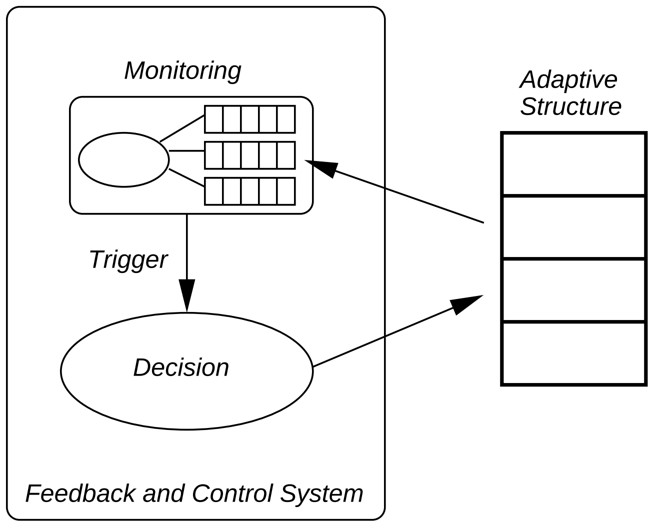

Figure 2. The elements of an adaptive processing system.

significant impact on how much the size of various hardware structures, such as caches and issue queues, impacts performance. Within some phases, large structures are beneficial, while in others, smaller structures may achieve almost the same performance.

To exploit application variability at the finer-grain level, a minimum of 10,000 cycles in duration, the mechanisms for monitoring and adapting the hardware must do so in a relatively rapid fashion, say 10-100 cycles for each of these minimum-length periods, to keep the time cost of adaptation reasonable (0.1-1%). This means that although coarser-grain adaptations may be performed in the operating system, monitoring and adaptation at a finer grain needs to be done at the hardware, the compiler, and/or the runtime system level. Most adaptive techniques have focused on exploiting fine-grain variations and these techniques are the subject of the rest of this article.

2 Elements of adaptive processing

Figure 2 shows the elements of a simple adaptive processing system, which consists of the adaptive hardware (one structure in this example) and a feedback and control system [1]. The overheads of the adaptive hardware must be nominal, and the feedback and control mechanisms kept simple, in order keep the costs of adaptive processing from rivaling its energy advantages.

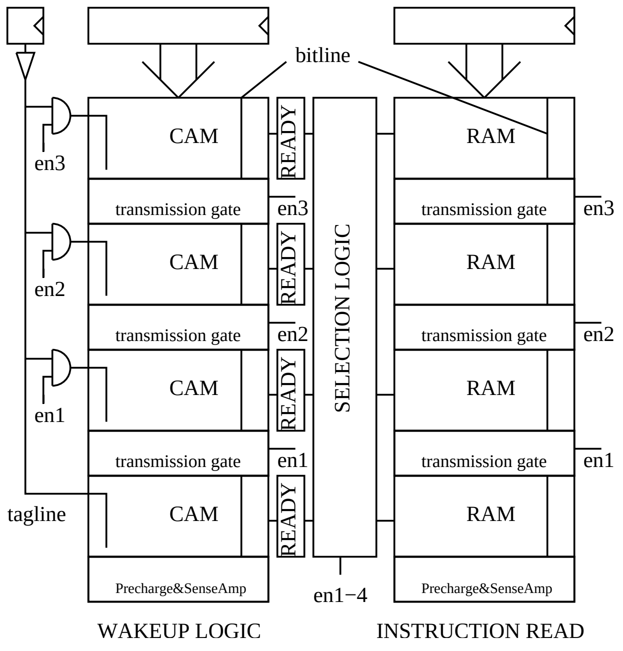

Figure 3. The organization of an adaptive issue queue [5].

2.1 Adaptive hardware structures

The adaptive hardware is organized such that its complexity (usually size) can be rapidly changed to fit current application needs. Regular, easily partitioned RAM and CAM-based structures, such as caches and issue queues, are widely used in modern processor design and their modular structure make them prime candidates for adaptation.

Two adaptive hardware structures are shown in Figures 3 and 4. Figure 3 shows a 32-entry adaptive issue queue which is partitioned into four equal size increments [5]. While the bottom increment (8 entries) is always enabled, the CAM and RAM arrays of upper increments are enabled on demand according to application needs. Transmission gates isolate the bitlines of the CAM and RAM increments, while simple gates achieve this function for the taglines. The Ready and Selection logic sections are partitioned as well, the latter by simply gating the select tree at the appropriate level depending on the number of enabled increments. Due to coarse level of adaptation, the quadratic relationship between wire length and delay, and the fact that the transmission gates serve to break the long bitlines into separate shorter wire sections, Buyuktosunoglu found that the use of such an adaptive structure would not impact clock frequency when used in a high-end processor. In addition, the gate count and energy overheads of this structure were found to be less than 3% [5].

An adaptive instruction cache, called the Dynamically Resizable I-Cache (DRI-Cache), is shown in Figure 4 [12].

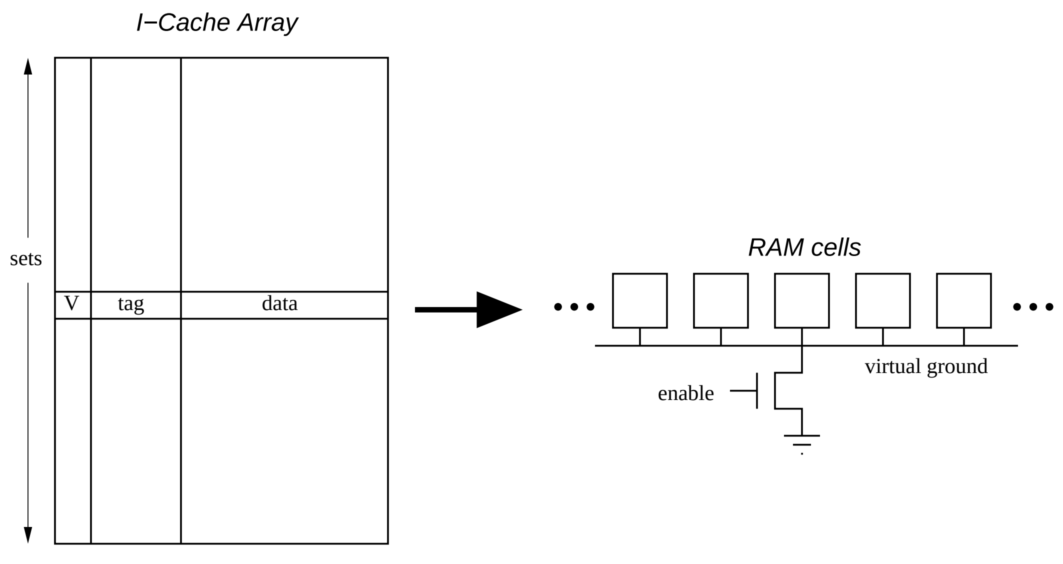

Figure 4. The Dynamically Resizable I-Cache (DRI-Cache) [12].

The cache is partitioned by sets such that associativity remains constant. A extra transistor is placed between the normal ground of the cells of a given set and the actual ground. The gate signal on this transistor enables the set; otherwise, it is disabled which virtually eliminates leakage in addition to dynamic power. Reducing both of these components is becoming paramount in advanced process technologies. Even with this relatively fine-grained *VDD-gating* approach, the area and speed overheads of the DRI-Cache are shown to be less than 8%, while when a set is disabled, its leakage current is reduced by 97%. One downside is that instructions are lost (and thus may need to be re-fetched) when a subarray is disabled. A recent modification proposed by the same group retains data within disabled cells while achieving much the same leakage gains [10]. This permits these disabled partitions to act as a backup to the enabled ones. In the event of a miss to the enabled partitions, the disabled ones are turned on and searched. This *backup* approach is used in Balasubramonian's adaptive memory hierarchy although only for dynamic power [3]. These two examples show that with careful engineering, adaptive hardware structures can be devised without unduly compromising area, speed, or energy.

2.2 Feedback and control system

The role of the feedback and control system (Figure 2) is to monitor the adaptive hardware and reconfigure it when appropriate. Because adaptive processing reduces the size of hardware structures, some increase in execution cycles is almost inevitable. The goal of the feedback and control system is to maximize energy savings while keeping performance loss below a specified level.

There are three major functions involved: monitoring the adaptive hardware, triggering a decision to be made, and making the decision. Both static and dynamic approaches can be used for each. Static approaches involving the compiler have a broader view of the entire program and permit simpler hardware, but they are limited by the static information available at compile time and require recompilation or binary rewriting of the application. On the other hand, dynamic approaches involving a combination of hardware and the runtime system have the advantages of having dynamic runtime information and the ability to run legacy applications but have a more limited program view and incur overhead. Each of these three feedback and control functions is discussed in turn with examples of static and dynamic approaches that have been used for each.

2.2.1 Monitoring

The monitoring system gathers statistics that infer the effectiveness of the adaptive hardware structures under control in order to guide reconfiguration decisions, or to aid in the identification of program phase changes. These statistics may be gathered each cycle or sampling may be used to reduce the monitoring overhead. For instance, in the adaptive issue queue proposed by Ponamarev [11], the number of valid queue entries is recorded at the end of a sample period and is averaged over a longer *interval* period. This average value is used to determine when to downsize the queue. At the same time, an overflow counter counts the number of cycles that dispatch is blocked due to a full queue and is used to guide upsizing decisions. These two statistics are simple to implement in hardware yet effective in guiding reconfiguration decisions. Folegnani and Gonzalez use program parallelism statistics, rather than issue queue utilization, to guide reconfiguration decisions [8]. Specifically, if instructions are rarely issuing from the back of the queue (that portion that holds the youngest instructions), then the queue is assumed to be larger than necessary and is downsized.

Cache miss rate is commonly used to guide cache configuration decisions. In the DRI-Cache for example, the average miss rate is measured over an interval of operation to determine whether the cache size needs to be changed. As miss rate information may already be available in microprocessor performance counters, this statistic is essentially available for free.

An alternative to hardware monitoring is compiler-based profiling. In this approach, the application is either instrumented and then run on the target machine to collect statistics, or is run on a detailed simulator with statistics gathered. Huang uses the latter to collect statistics on the execution length of subroutines for phase detection [9]. In both cases, the application behavior observed during the profiling run must be representative of that encountered in production. Since this cannot be assumed for many general-purpose applications, and cheap hardware counters are readily available in modern microprocessors or can be added with modest overhead, hardware-based monitoring is more frequently used in adaptive processing.

2.2.2 Triggering

There are several approaches used to trigger that a reconfiguration decision should be made. The first approach is to trigger based on particular characteristics of the monitored statistics. For instance, Ponamarev's adaptive issue queue scheme upsizes the queue when the average number of valid queue entries over the interval period is such that this average number of instructions could have been held in a smaller configuration. This is easily determined from the sampled average occupancy statistics, the current size of the queue, and the possible queue configurations. Alternatively, the queue is immediately upsized when the overflow counter exceeds a threshold to prevent non-negligible performance loss. An alternative approach is to use detection of phase changes to trigger reconfiguration decisions. In the adaptive memory hierarchy proposed by Balasubramonian [3], cache miss rates and branch counts of the last two intervals are compared. If a significant change in either is detected, a phase change is assumed to have occurred. Dhodapkar and Smith [6] improve on this approach by triggering based on differences in *working set signatures*. A working set signature is a compact approximation of a *working set*, which represents the set of distinct memory elements touched over some period. A significant difference in working set signatures constitutes a phase change.

Positional adaptation as described by Huang [9] uses program structure to identify major program phases. Specifically, either compile-time or run-time profiling is used to determine long-running subroutines for which the appropriate configuration should be selected. In the static approach, a profiling run measures the total execution time and the average execution time per invocation of each subroutine. Phases are identified as subroutines with values for these quantities that exceed pre-set thresholds. The entry and exit points of these routines are instrumented to trigger that a reconfiguration decision be made.

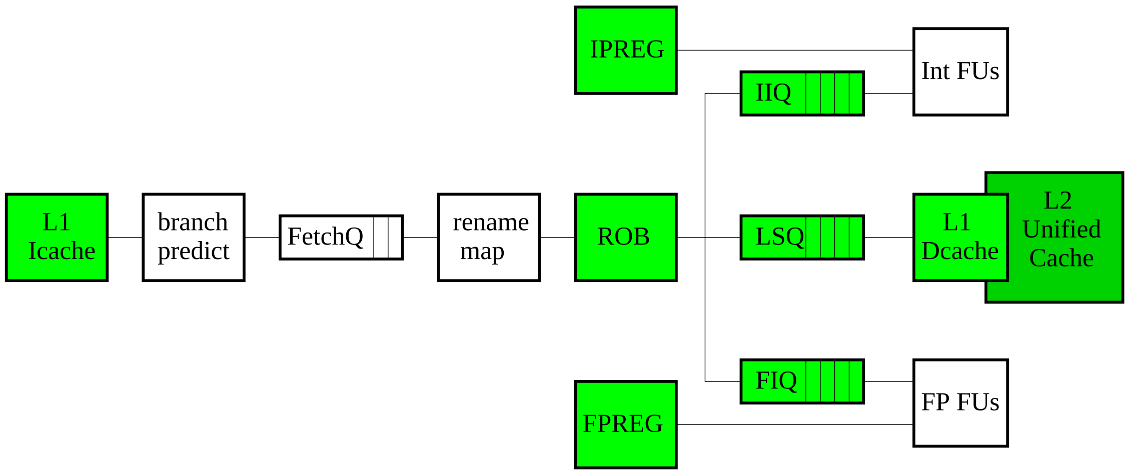

Figure 5. A four-way superscalar design in which the shaded elements are adaptive [7].

2.2.3 Making a decision

Once a trigger event has occurred, a decision must be made regarding the adaptive configuration that should be used over the next period of operation. If the number of configuration options is small, then a simple trial-and-error approach can be used: each configuration can be tried over consecutive intervals and the best-performing one selected. This *exploration* approach is used in Balasubramonian's adaptive memory hierarchy [3]. A second approach is to use the monitored statistics to make a decision. When upsizing the adaptive issue queue, Ponamarev chooses a new configuration based on the difference between the current number of queue entries and the sampled average of the number of valid entries over the interval period. If this difference is large, then the queue may be downsized more aggressively [11]. The underlying assumption of these approaches is that the current behavior is indicative of that to come. If the rate of behavioral change rivals the evaluation period, then significant error can occur.

Decision prediction attempts to circumvent this problem by using past configuration information to predict the best-performing option in the future. Dhodapkar and Smith save their working set signatures along with configuration information in a RAM. When the current signature closely matches one stored in the RAM, the configuration is looked up and used immediately [6].

3 An example adaptive processing system

Figure 5 shows an example adaptive processing system in which the issue queues, load/store queue, Reorder Buffer, register files, and caches of a four-way dynamic superscalar processor are all adaptive [7]. With multiple adaptive structures, the explosion in the number of possible combinations of configurations creates two main challenges. First, exploration cannot be used as its overhead for so many options would be prohibitive. Second, configuring multiple structures creates a challenging cause-and-effect assignment problem. Precisely, it is difficult to know whether a change in application performance is due to a change in program behavior, reconfiguring a different hardware structure, or reconfiguring this particular one. For these reasons, instructions per cycle performance is avoided as a monitoring statistic and local statistics that accurately infer changes in the particular structure's behavior are used.

For the caches, the backup approach of Balasubramonian is used, but the control approach is different. Statistics are gathered that permit the performance and energy of *all possible* cache configurations to be determined. Figure 6 shows the L1 data cache and the fundamental operations performed when the data is not present in the cache. The most-recently-used (MRU) state of the cache ways (four in this example) is maintained and an MRU counter is associated with each of the four states. The primary part (shown in white) is accessed, and upon a miss, the backup part (shaded) is accessed which also results in a miss. The replaced block in the primary section is written to the backup section. The backup block is written back if dirty and discarded otherwise. The miss counter is also incremented.

A hit within either the primary or backup part results in an update of the MRU state and counters as shown in Figure 7. In the example, block A is the most recently used block (MRU[0]), B is the second most recently used block (MRU[1]), and so on. When a block is accessed, the counter associated with that block's MRU state is incremented, and the MRU state is updated accordingly. For example, the access of block B increments the counter for the second most recently used block (MRU[1]). Block B is now the most recently used block and A the second. An access to C increments MRU[2] and changes the MRU state. The next access to C increments MRU[0] (since it was just accessed and so is the most recently used block) while the access to D increments MRU[3]. At the end of an interval, the MRU counters and the miss counter are read by a runtime routine and used to calculate the number of primary and backup hits and overall misses that would have occurred for each configuration during that interval. Based on the time and energy costs of hits and misses for each configuration, the best configuration is chosen for the next interval.

The MRU counters also permit the calculation of the actual performance loss incurred by the reconfigurable cache compared to some fixed baseline. If that baseline is a subset of the reconfigurable cache, then the MRU

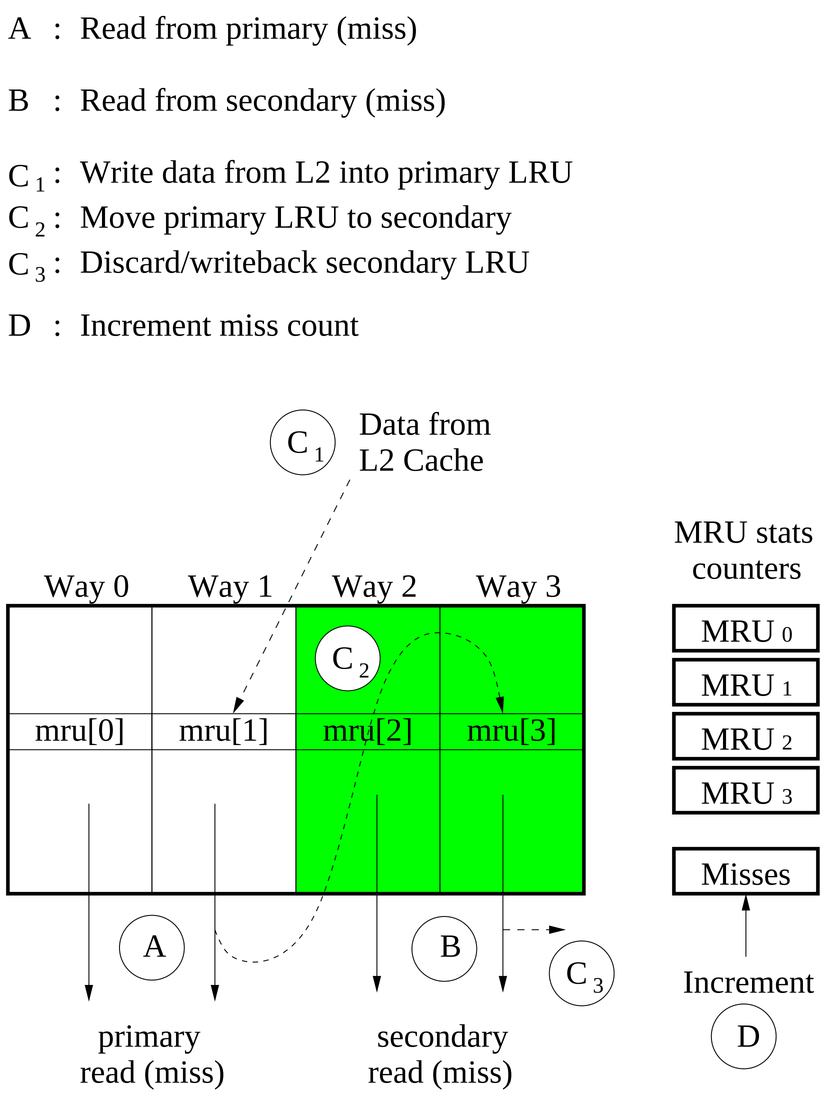

Figure 6. L1 data cache operations on a miss to both the primary and backup (secondary) sections.

counters can be used to determine what its performance would have been if it was used. This value can be compared against that of the reconfigurable cache to keep a running total of the performance loss up to this point in time. This information can be compared against the target performance loss to determine if the reconfigurable cache is doing better or worse than the target. If the loss is less than the target, then the controller can be more aggressive in trading performance for energy. If the performance loss is excessive, the controller is more conservative in an attempt to reduce the overall loss. This “accounting” operation permits a tight bound on the performance loss while maximizing energy savings.

For the queues, register files, and Reorder Buffer, the physical structure is similar to that of Buyuktosunoglu and a variation of the Ponomarev feedback and control approach is used. However, an interesting aspect of adaptive register files is that there is no *a priori* way to determine without compiler help that a register value will not be

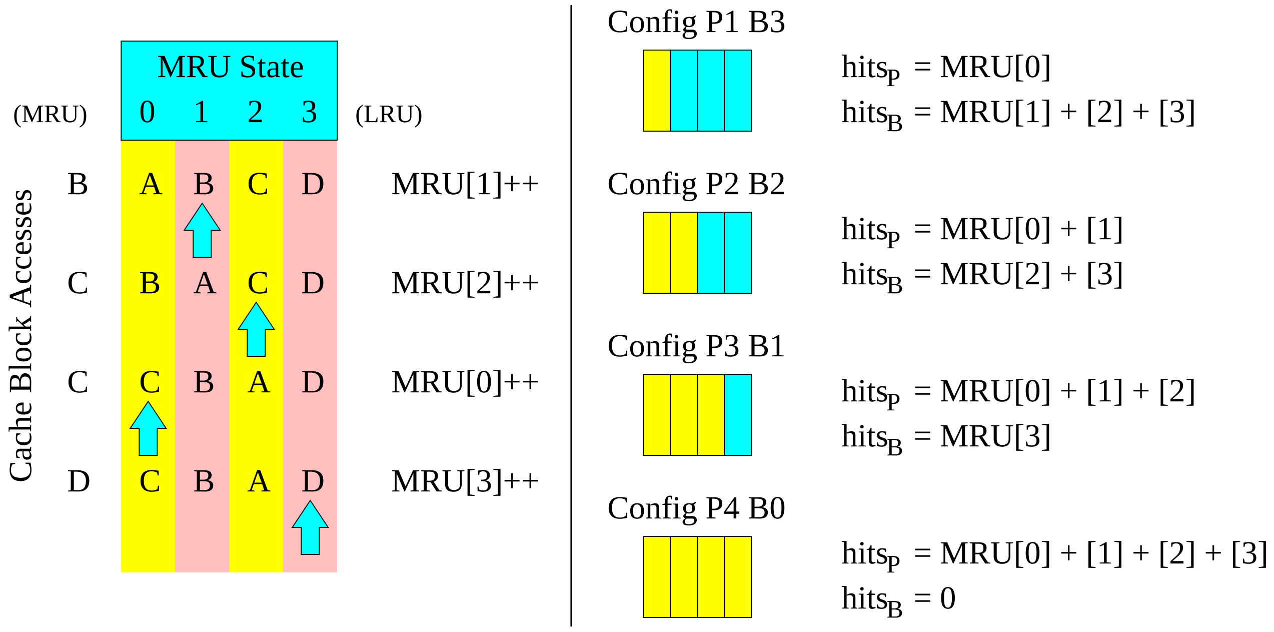

Figure 7. MRU state changes and counter updates when accessing four different cache blocks (left). The calculations performed to determine the number of primary and backup hits for each configuration (right). A configuration denoted as *Px By* has *x* primary and *y* backup partitions.

used in the future. Even with compiler support, there would be only a small fraction of registers that would be able to be disabled in many appplication if such a requirement was imposed. Thus, when downsizing the register file, the contents of all active physical registers must be preserved. The solution is to move the values of registers in the partition to be disabled into an active partition. Fortunately, this can be largely achieved with the existing renaming mechanism found in modern machines. First, any physical register from the to-be-disabled partition is prevented from being allocated to newly renamed instructions. Then a small runtime routine that performs the instruction *move rx*, *rx* for each logical register *rx* is executed. This causes any logical register values stored in physical registers in the to-be-disabled partition to be read and transferred to a newly allocated physical register from the enabled part of the register file. The hardware performs the transfer as part of normal instruction processing (care must be taken that the implementation does not equate the *move rx*, *rx* instruction with a no-op) and the rename table is updated with the new mapping as part of the normal rename mechanism, permitting the partition to then be disabled.

Figure 8 summarizes the energy saved within the adaptive structures as well as the overall performance degradation for a combination of three Olden, seven SPEC integer, and four SPEC floating point benchmarks. These are

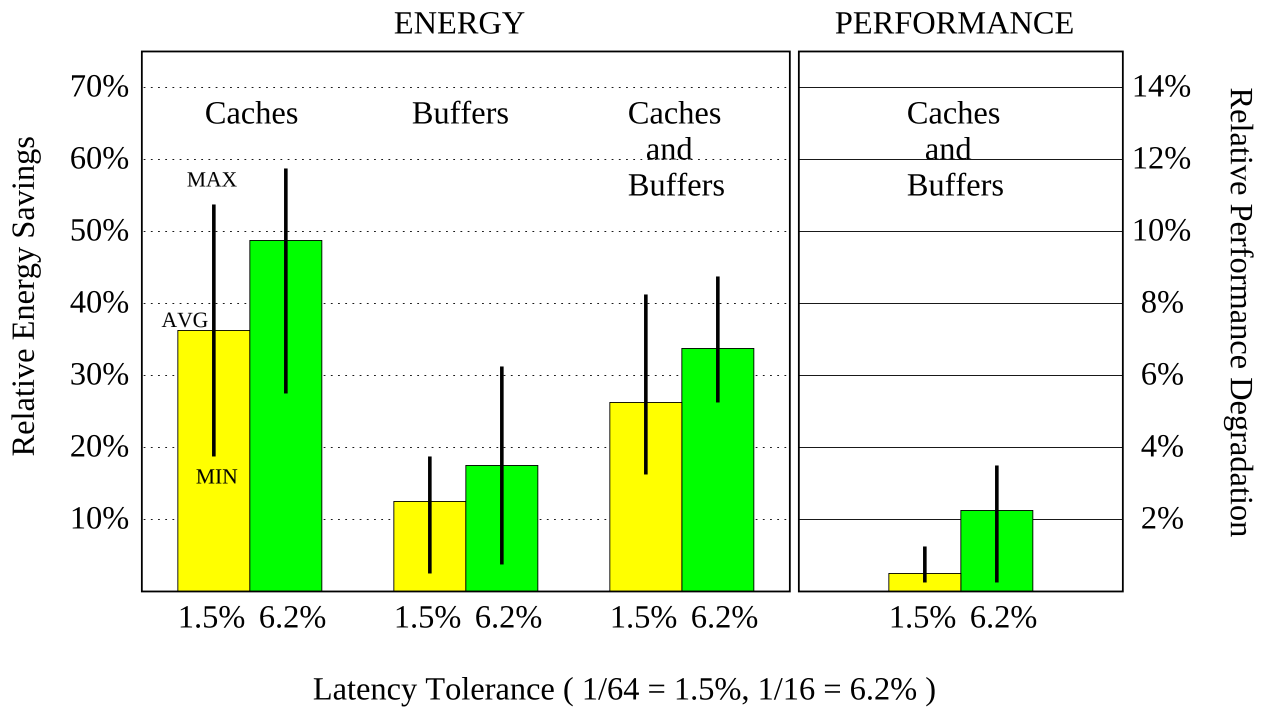

Figure 8. Energy savings for the caches only, buffers (queues, register files, and Reorder Buffer) only, and for both combined, as well as overall performance degradation. "Error bars" show the range of values for the tested benchmarks.

plotted as a function of the target performance degradation that the adaptive processing system can allow for. The 1.5% and 6.2% values correspond to the power-of-two fractions 1/64 and 1/16, respectively, which simplifies the calculations. As expected, a higher target performance degradation permits more energy to be saved. In addition, due to the ability of modern caches to hide latency with other work, the actual performance degradation is much less than the target. The higher energy savings achieved in the caches is primarily because these structures dissipate greater energy overall. For the 1.5% performance degradation target, a 28% energy savings in the adaptive structures is achieved with a 0.6% actual overall performance degradation. For the 6.2% target, the savings is 34% with a 2.1% performance loss.

The energy savings needs to be calculated for the processor as a whole in order to assess the cost/savings of adaptive processing. From [4], the issue queues, Reorder Buffer, caches, and register files of a modern superscalar processor can easily consume over 50% of the total chip power. Using 50% as a conservative scaling factor for the energy results, the adaptive processing system can achieve roughly a 14% overall chip energy savings for a 0.6% performance loss, or 17% for the larger 2.1% loss. These results compare favorably with the 3:1 power savings to performance degradation ratio typically achieved with voltage scaling.

4 Looking ahead

Many of the adaptive techniques explored thus far have focused exclusively on reducing dynamic power. (The DRI-Cache work is a notable exception.) In future process technologies, leakage power is expected to rival dynamic power in magnitude. Adaptive techniques are poised to address both leakage and dynamic power. In the system of Figure 5, a technique like VDD-gating can be directly applied to the issue queues, Reorder Buffer, and register files (since disabled partitions are ensured to be empty before being turned off) as well as in the L1 I-Cache. An approach like [10] can be used in the L1 D-cache and L2 cache in order to preserve their state.

An important consideration is that the extra transistors added for adaptive processing (each of which will contribute to overall leakage) is small relative to the energy saved. In addition, the decision logic is invoked infrequently and therefore can itself be placed into a low leakage state until a trigger event occurs. All of these characteristics make adaptive processing a promising technology for saving both dynamic and leakage energy in future process technologies.

References

[1] D.H. Albonesi. Dynamic IPC/clock rate optimization. *Proceedings of the 25th International Symposium on Computer Architecture*, pages 282–292, June 1998.

[2] D.H. Albonesi. The inherent energy efficiency of complexity-adaptive processors. *Proceedings of the 1998 Power-Driven Microarchitecture Workshop*, pages 107–112, June 1998.

[3] R. Balasubramonian, D.H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. *Proceedings of the 33rd International Symposium on Microarchitecture*, pages 245–257, December 2000.

[4] P. Bose et al. Early-stage definition of LPX: A low power issue-execute processor. *Workshop on Power-Aware Computer Systems*, February 2002.

[5] A. Buyuktosunoglu, S. Schuster, D. Brooks, P. Bose, P. Cook, and D.H. Albonesi. A circuit level implementation of an adaptive issue queue for power-aware microprocessors. *Proceedings of the 11th Great Lakes Symposium on VLSI*, March 2001.

[6] A.S. Dhodapkar and J.E. Smith. Managing multi-configuration hardware via dynamic working set analysis. *International Symposium on Computer Architecture*, May 2002.

[7] S. Dropsho, A. Buyuktosunoglu, R. Balasubramonian, D.H. Albonesi, S. Dwarkadas, G. Semeraro, G. Magklis, and M.L. Scott. Integrating adaptive on-chip storage structures for reduced dynamic power. *Proceedings of the 11th International Conference on Parallel Architectures and Compilation Techniques*, pages 141–152, September 2002.

[8] D. Folegnani and A. Gonzalez. Energy-effective issue logic. *International Symposium on Computer Architecture*, June 2001.

[9] M.C. Huang et al. Positional adaptation of processors: application to energy reduction. *International Symposium on Computer Architecture*, June 2003.

[10] H. Kim and K. Roy. Dynamic Vt SRAMs for low leakage. *International Symposium on Low Power Electronics and Design*, August 2002.

[11] D. Ponomarev et al. Dynamic allocation of datapath resources for low power. *International Symposium on Microarchitecture*, December 2001.

[12] M.D. Powell et al. An energy-efficient high-performance deep-submicron instruction cache. *IEEE Transactions on VLSI Systems*, February 2001.

[13] B. Xu and D.H. Albonesi. A methodology for the analysis of dynamic application parallelism and its application to reconfigurable computing. *Proceedings of the SPIE International Symposium on Reconfigurable Technology: FPGAs for Computing and Applications*, pages 78–86, September 1999.