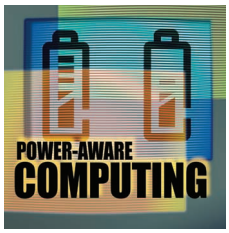
Dynamically Tuning Processor Resources with Adaptive Processing



Using adaptive processing to dynamically tune major microprocessor resources, developers can achieve greater energy efficiency with reasonable hardware and software overhead while avoiding undue performance loss.

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
University of Rochester

Pradip Bose
Alper Buyuktosunoglu
Peter W. Cook
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 proliferate rapidly throughout society, while battery technology improves at a much slower pace. In large data centers that handle everything from online orders for a dot-com company to sophisticated Web searches, row upon row of tightly packed computers may be warehoused in a city block. Microprocessor energy wastage in such a facility directly translates into higher electric bills. Simply receiving sufficient electricity from utilities to power such a center is no longer certain. Given this situation, energy efficiency has rapidly moved to the forefront of modern microprocessor design.

The *adaptive processing* approach to improving microprocessor energy efficiency dynamically tunes major microprocessor resources—such as caches and hardware queues—during execution to better match varying application needs.[1,2] This tuning usually involves reducing the size of a resource when its full capabilities are not needed, then restoring the disabled portions when they are needed again. Dynamically tailoring processor resources in active use contrasts sharply with techniques that simply turn off entire sections of a processor when they become idle. Presenting the application with the required amount of hardware—and nothing more—throughout its execution can achieve a potentially significant reduction in energy consumption.

The challenges facing adaptive processing lie in achieving this greater efficiency with reasonable hardware and software overhead, and doing so without incurring undue performance loss. Unlike reconfigurable computing, which typically uses very different technology such as FPGAs, adaptive processing exploits the dynamic superscalar design approach that developers have used successfully in many generations of general-purpose processors. Whereas reconfigurable processors must demonstrate performance or energy savings large enough to overcome very large clock frequency and circuit density disadvantages, adaptive processors typically have baseline overheads of only a few percent.

VARYING APPLICATION BEHAVIOR

Application needs for particular hardware resources—such as caches, issue queues, and instruction fetch logic within a dynamic superscalar processor—can vary significantly from application to application and even within the different phases of a given application. Several studies have demonstrated this dynamic application behavior. Figure 1 shows the execution behavior of the SPEC95 ijpeg application in an early study.[3] The graphs show the

0018-9162/03/$17.00 © 2003 IEEE Published by the IEEE Computer Society

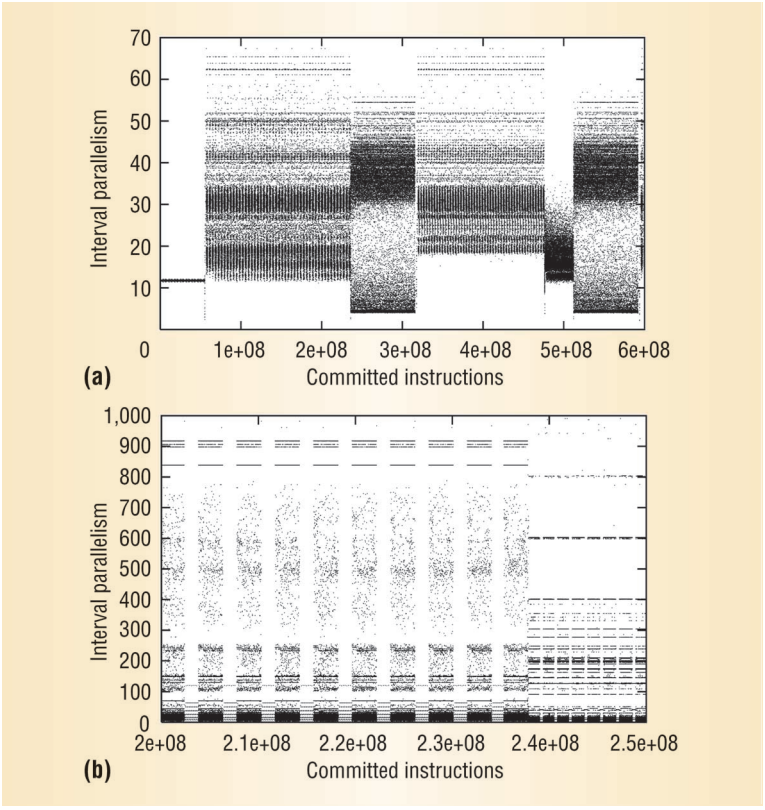


Figure 1. Interval parallelism for the SPEC95 benchmark* ijpeg*. Each dot in the graph is the ratio of total committed instructions to total cycles for an interval of (a) 2,000 instructions and (b) 100 instructions.

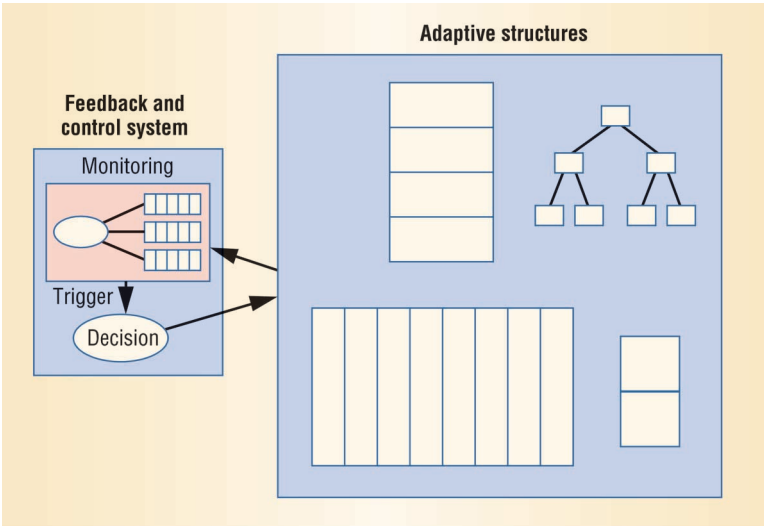


Figure 2. Elements of an adaptive processing system. The system adapts modular hardware structures and uses simple feedback and control to reduce hardware overhead.

interval parallelism, expressed as total committed instructions over total cycles, achieved with every set number of instructions. Each dot in Figure 1a equals the interval parallelism for 2,000 instructions, while Figure 1b shows the interval parallelism for 100 instructions. To isolate the limits to parallelism to primarily those inherent in the application, the machine being modeled has almost infinite resources and perfect caches.

These graphs show multiple levels of phases. Figure 1a demonstrates major phase shifts occurring at the granularity of millions of instructions. Figure 1b shows that within these major phases, finer-grained phases last for up to tens of thousands of instructions. Within these phases, parallelism varies widely, and this variation affects the degree to which the size of various hardware structures, such as caches and issue queues, affects performance.[3] Within some phases, large structures provide clear benefits, while in others, much smaller structures can achieve almost the same performance while saving significant energy.

To exploit application variability at a finer-grained level—at least 10,000 cycles in duration—hardware monitoring and control must occur relatively rapidly, on the order of 10 to 100 cycles, to achieve the acceptable time overhead for adaptation in the range of 0.1 to 1 percent. This means that although developers can perform coarser-grained adaptations through the operating system, they must incorporate finer-grained adaptations at the hardware, compiler, and runtime system levels.

ADAPTIVE PROCESSING ELEMENTS

Figure 2 shows the elements of an adaptive processing system. To prevent the costs of adaptive processing from rivaling its energy advantages, the feedback and control mechanisms must be kept simple and the adaptive hardware overhead must be nominal.[1,2]

Adaptive hardware structures

Chip designers organize adaptive hardware so that they can rapidly change its complexity, usually a function of its size or width, to fit current application needs. Modern processor designs feature easily adaptable modular structures, such as caches and issue queues.

Figure 3 shows a 32-entry adaptive-issue queue, partitioned into four equal increments.[4] Although the system always enables the bottom increment of eight entries, it enables the upper increments' content-addressable memory and random-access memory arrays on demand, according to the application's needs. Transmission gates isolate the CAM and RAM bitlines, while simple gates achieve this function for the taglines.

The designer also partitions the wake-up and selection logic sections, the latter by simply gating the select tree at the appropriate level depending on the number of enabled increments. Given the

coarse level of adaptation, Alper Buyuktosunoglu and colleagues discovered that using such an adaptive structure in a high-end processor would not affect clock frequency.[4] The gate count and energy overhead each increased by less than 3 percent.

Figure 4 shows one of the first proposed adaptive instruction caches, the dynamically resizable I-cache.[5] The DRI-cache can disable individual cache sets, thereby maintaining constant associativity. To enable or disable the set, the DRI-cache uses an extra high-threshold-voltage transistor for each row of RAM cells—which corresponds to one cache set.

The transistor is placed between the cells' normal ground for an individual set and the actual ground. A logic high on the transistor gate signal enables this set. Otherwise, the set is disabled, which significantly reduces its leakage current and dynamic power. Despite this relatively fine-grained voltage-gating approach, the DRI-cache's area and speed overhead are less than 8 percent. Disabling a set reduces its leakage current by 97 percent, but the contents are lost. A recently proposed alternative[6] disables part of the cache by dynamically switching to a lower voltage, thereby retaining cache contents while achieving about the same reduction in leakage power.

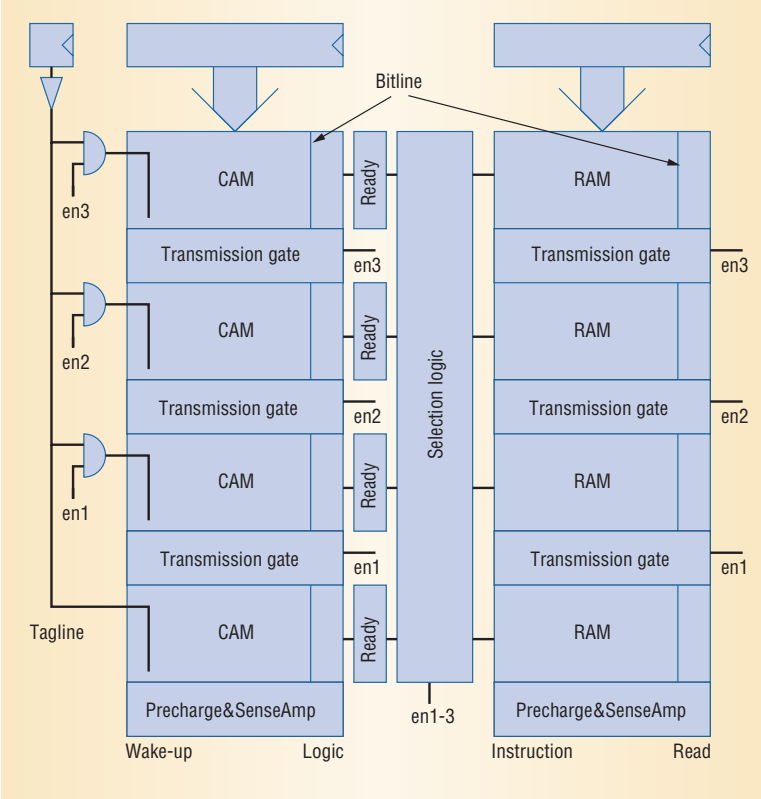


Figure 3. Adaptive-issue queue. The 32-entry queue is partitioned into four equal increments.

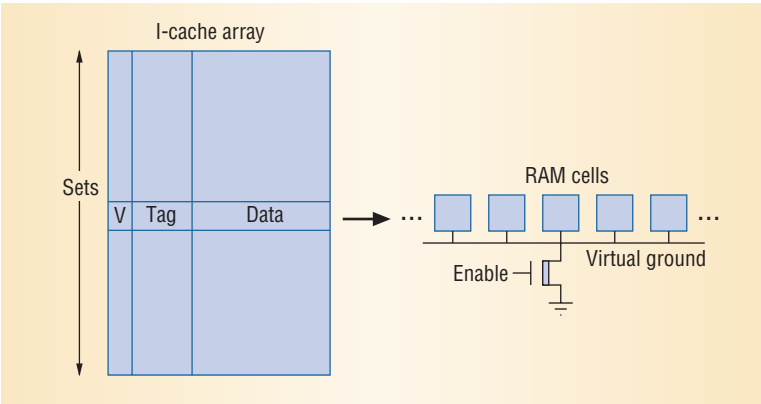


Figure 4. Dynamically resizable I-cache. The DRI-cache can disable individual cache sets, thereby maintaining constant associativity.

Feedback and control system

Because adaptive processing reduces the effective size of hardware structures, some increase in execution cycles is almost inevitable. Thus, the feedback and control system seeks to maximize the savings in energy while limiting any performance loss below a specified level. This involves three major functions: monitoring the adaptive hardware, triggering a decision, and making the decision.

Designers use either static or dynamic approaches for each of these functions. Static approaches involving the compiler take a broader view of the entire program and permit simpler hardware. However, the static information available at compile time limits these approaches and requires recompiling the application or performing a binary rewrite. Alternatively, dynamic approaches that involve a combination of hardware and the runtime system offer the advantages of providing dynamic runtime information and the ability to run legacy applications. However, they suffer from a more limited program view and incur overhead.

Monitoring. To guide reconfiguration decisions, the monitoring system gathers statistics that infer the effectiveness of the adaptive hardware structures the processor is controlling or that help identify program phase changes. The system can gather these statistics in hardware each cycle, or it can use sampling to reduce monitoring overhead.

Dmitry Ponomarev and colleagues[7] proposed an adaptive-issue queue that records the number of valid queue entries at the end of a sample period and averages them over a longer interval period. The system uses this average value to determine when to downsize the queue. Simultaneously, an overflow counter tracks the number of cycles whose dispatch is blocked due to a full queue. The system uses this result to guide upsizing decisions. Although

Managing Multiple Low-Power Adaptation Techniques: The Positional Approach

Michael C. Huang, University of Rochester
Jose Renau and Josep Torrellas, University of Illinois at Urbana-Champaign

A major challenge in adaptive processing is to determine when to trigger an adaptation. To do this, we need to partition the program into phases that behave differently enough to warrant adaptation. Ideally, the behavior within each phase is homogeneous and predictable. The granularity of each phase should not be too fine or too coarse. If it is too fine, transient states and adaptation overheads can negate any gains. If it is too coarse, the behavior probably is not homogeneous.

In the context of improving energy efficiency, we use *low-power techniques* for adaptation. An LPT is a hardware structure that, if activated, typically saves energy at the expense of some performance. The processor activates LPTs at the beginning of a phase on the basis of their predicted effect.

We classify adaptation approaches based on how they exploit program behavior repetition. The conventional *temporal approach*[1] exploits the similarity between successive intervals of code in dynamic order; the newer *positional approach*[2] exploits the similarity between different invocations of the same code section.

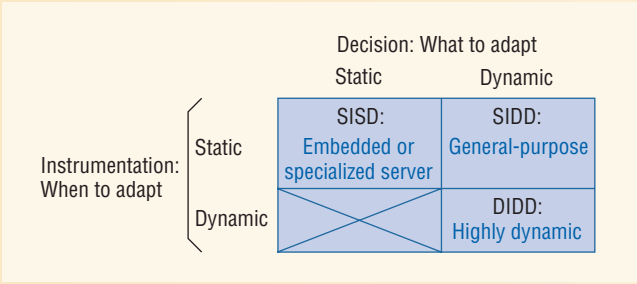


Figure A. Different implementations of positional adaptation and targeted workload environments.

The two approaches activate and deactivate LPTs based on different criteria. Specifically, temporal schemes divide the execution into time intervals and predict the upcoming interval's behavior based on previous intervals' behavior. Positional schemes, instead, associate program behavior with a particular code section. Thus, a positional scheme tests LPTs on different executions of the same code section. Once the positional scheme determines the best configuration, it applies that configuration on future executions of the same code section. This approach is based on the intuition that program behavior is largely determined by the code being executed. Experimental analysis shows that calibration is more accurate in the positional approach.[2]

Positional adaptation is also very flexible. We propose three

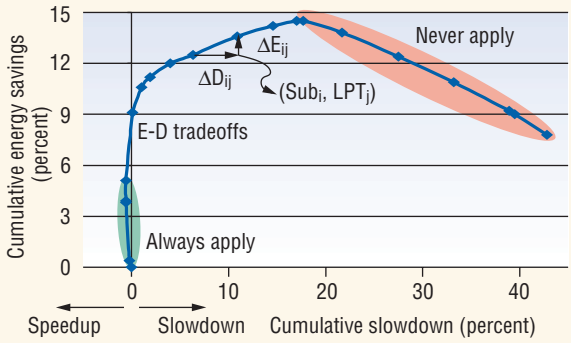


Figure B. Energy-delay tradeoff curve. Starting from left to right in the E-D tradeoffs region, the positional scheme applies pairs until the cumulative slowdown reaches the slack.

simple to implement in hardware, these two statistics can effectively guide reconfiguration decisions.

Daniele Folegnani and Antonio Gonzalez use program parallelism statistics, rather than issue queue usage, to guide reconfiguration decisions.[8] Specifically, if the processor rarely issues instructions from the back of the queue—that portion of the queue that holds the most recent instructions—the system assumes the queue to be larger than necessary and downsizes it. The system also periodically upsizes the queue at regular intervals to limit performance loss.

Researchers commonly use cache miss rate to guide cache configuration decisions. The DRI-cache, for example, measures the average miss rate over a set operation interval to determine whether to change the cache size. As miss rate information may already be available in microprocessor performance counters, the system can essentially acquire this statistic for free.

Compiler-based profiling offers an alternative to hardware monitoring. With this approach, developers either instrument the application and run it on the target machine to collect statistics, or they run it on a detailed simulator to gather the statistics. Michael Huang and his colleagues use the simulator approach to collect statistics about the execution length of subroutines for phase detection.[9]

The application behavior observed during the profiling run must be representative of the behavior encountered in production. Because this assumption may not hold for many general-purpose applications, and inexpensive 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.

Triggering. A microprocessor can use several approaches to trigger a reconfiguration decision. The first approach reacts to particular characteristics of the monitored statistics. For example, Ponomarev's adaptive-issue queue scheme upsizes the queue when the average number of valid queue entries over the interval period is low enough that

different implementations that target different workload environments. They differ on which adaptation decisions they make statically and which decisions they make at runtime.[2] Specifically, as Figure A shows, *instrumentation* (I) is the selection of when to adapt the processor, and *decision* (D) is the selection of what LPTs to activate or deactivate at that time. The system can make each selection *statically* (S) before execution or *dynamically* (D) at runtime. For example, an implementation can produce static instrumentation and static decision (SISD).

The targeted environments are labeled as embedded or specialized server, general-purpose, and highly dynamic. In these implementations, we use the program's major subroutines as the code section. The core control algorithm for all the implementations is essentially the same.

Tests of the different LPTs on different subroutines record the impact on energy and performance for comparison with other LPT and subroutine combinations. Specifically, we rank the pairs in decreasing order of energy savings per unit slowdown.

Figure B shows this ranking for a sample application in a system with several LPTs. The difference between the three implementations is how much information they provide. The more static schemes are more accurate because they have more information thanks to offline profiling.

The origin in the figure corresponds to the system with no activated LPT. As we follow the curve, we add the contribution of all subroutine-LPT pairs from most to least efficient, accumulating energy reduction (*y*-axis) and execution slowdown (*x*-axis). As an example, Figure B shows the contribution of a pair (subroutine$_i$, LPT$_j$) that saves ΔE_{ij} and slows down the program ΔD_{ij}.

We divide the curve into three main regions based on the results for each pair: improving both performance and energy (*Always apply*), saving energy at the cost of performance degradation (*E-D tradeoffs*), and degrading both energy and performance (*Never apply*). As the names suggest, we apply all the pairs in the first region and no pairs in the third region. Given a slack—or tolerable performance degradation—we start from left to right in the E-D tradeoffs region and apply pairs until the cumulative slowdown reaches the slack. In two experiments, the positional schemes boosted the energy savings by an average of 84 percent and 50 percent over several temporal schemes.[2]

References

1. R. Balasubramonian et al., "Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures," *Proc. Int'l Symp. Microarchitecture*, IEEE CS Press, 2000, pp. 245-257.
2. M. Huang, J. Renau, and J. Torrellas, "Positional Adaptation of Processors: Application to Energy Reduction," *Proc. Int'l Symp. Computer Architecture*, IEEE CS Press, 2003, pp. 157-168.

Michael C. Huang *is an assistant professor in the Department of Electrical and Computer Engineering, University of Rochester. Contact him at michael.huang@ece.rochester.edu.*

Jose Renau *is a PhD candidate in the Computer Science Department at the University of Illinois at Urbana-Champaign. Contact him at renau@cs.uiuc.edu.*

Josep Torrellas *is a professor and Willett Faculty Scholar in the Computer Science Department at the University of Illinois at Urbana-Champaign. Contact him at torrellas@cs.uiuc.edu.*

a smaller configuration could have held the average number of instructions. The system can easily determine this condition from the sampled average occupancy statistics, the queue's current size, and the possible queue configurations. To prevent nonnegligible performance loss, the system upsizes the queue immediately when the overflow counter exceeds a preset threshold.

Another approach detects phase changes to trigger reconfiguration decisions. Balasubramonian's adaptive memory hierarchy[10] compares cache miss rates and the branch counts of the last two intervals. If the system detects a significant change in either, it assumes that a phase change has occurred. Ashutosh S. Dhodapkar and James E. Smith[11] improve on this approach by triggering a phase change in response to differences in working-set signatures—compact approximations that represent the set of distinct memory elements accessed over a given period. A significant difference in working-set signatures constitutes a phase change.

Still another approach triggers a resource upsizing only when a large enough increase in performance would be expected. This technique can be used to upsize an adaptive-issue queue.[12] A larger instruction window permits the stall time of instructions waiting in the window to be overlapped with the execution of additional ready instructions in the larger window. However, if this overlap time is not sufficiently large, upsizing the queue will provide little performance benefit. The system estimates the overlap time and uses it to trigger upsizing decisions.

Huang and colleagues proposed positional adaptation,[9] which uses the program structure to identify major program phases. Specifically, as the "Managing Multiple Low-Power Adaptation Techniques: The Positional Approach" sidebar describes, this approach uses either compile-time or runtime profiling to select an appropriate configuration for long-running subroutines. In the static approach, a profiling run measures the total execution time and the average execution time per invocation of each subroutine. Developers identify phases as subroutines with values for those quan-

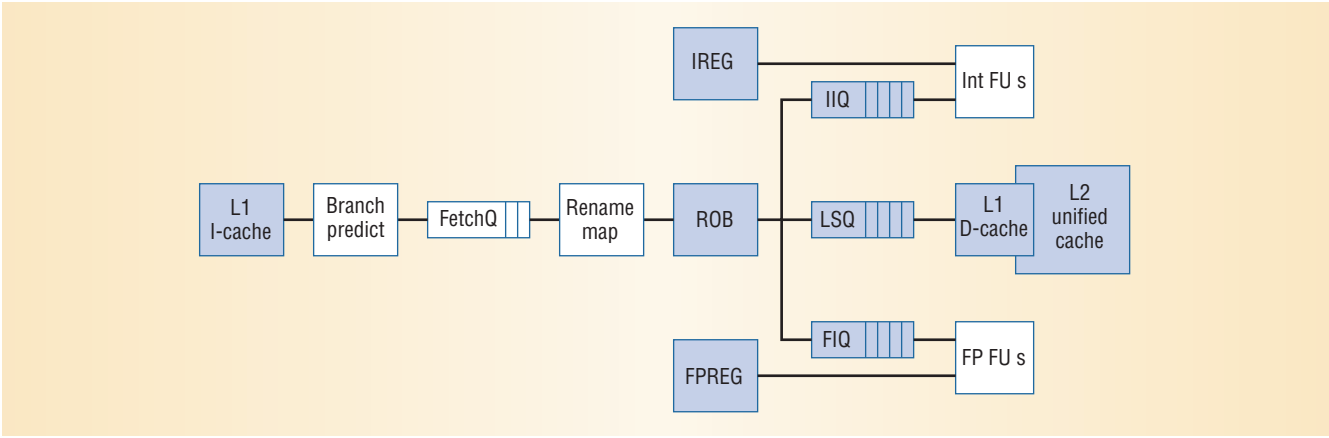


Figure 5. Adaptive processing system. The shaded elements—the issue queues, load and store queue, reorder buffer, register files, and caches—have been redesigned for adaptive processing.

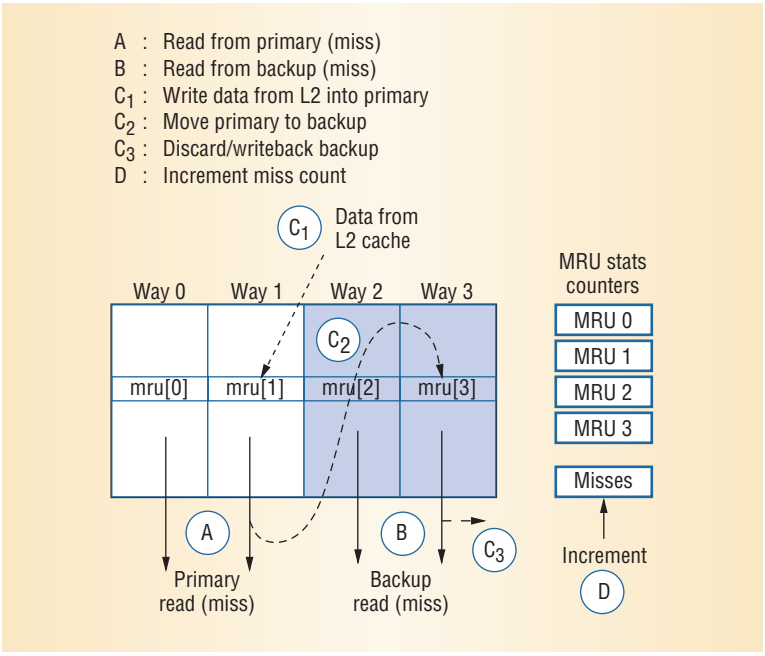


Figure 6. L1 data cache operations. When data arrives from the L2 cache, on a miss to both the primary and backup sections, the system writes the replaced block in the primary section to the backup section and increments the miss counter.

tities that exceed preset thresholds, then they instrument the entry and exit points of these routines to trigger a reconfiguration decision.

Making a decision. Once a trigger event occurs, the system must select the adaptive configuration to use over the next operational period. If the number of configuration options is small, the system can use a simple trial-and-error approach: It can try each configuration over consecutive intervals and select the best-performing configuration. Balasubramonian's adaptive memory hierarchy uses this exploration approach.[10]

A second approach uses the monitored statistics to make a decision. Ponomarev's adaptive-issue queue chooses a new configuration based on the difference between the current number of queue entries and the sampled average of valid entries over the interval period. If this difference is large, the system can downsize the queue more aggressively.[7]

These approaches operate from the underlying assumption that current behavior indicates future behavior. If the behavioral change rate rivals that of the evaluation period, significant error can occur. Decision prediction attempts to circumvent this issue by using past configuration information to predict the best-performing option. Dhodapkar and Smith[11] save their working-set signatures in RAM, along with configuration information. When the current signature closely matches a signature stored in RAM, the system looks up the configuration and immediately uses it. Similarly, in Huang's positional scheme, for every code section, once the best configuration is determined, it is remembered and applied to all future executions of that code section.

SAMPLE ADAPTIVE PROCESSING SYSTEM

Figure 5 shows a sample system in which adaptive processing has been applied to the issue queues, load and store queue, reorder buffer, register files, and caches of a four-way dynamic superscalar processor.[13] Equipped with these multiple adaptive structures, the large number of possible configuration combinations creates two primary challenges.

First, exploration is not an option because the overhead would be prohibitive. Second, configuring multiple structures creates a challenging cause-and-effect assignment problem, making it difficult to know whether a change in application performance stems from a change in program behavior, reconfiguring a different hardware structure, or reconfiguring this particular structure. For these reasons, rather than using instructions-per-cycle performance as a monitoring statistic, we use local statistics that more accurately infer changes in a particular structure's behavior.

For the caches, we use Balasubramonian's backup approach. First, our system accesses the

enabled partitions; it accesses the disabled partitions only on a miss. To determine which partitions we want the system to enable initially, we gather statistics to determine the performance and energy of all possible cache configurations during a particular interval of operation.

Figure 6 shows the fundamental operations the system performs when data is missing from the L1 data cache. The system maintains the cache's most recently used state and associates an MRU counter with each of the four states. The system accesses the primary part of the MRU, shown in white, first. Upon a miss, the system accesses the backup part, shown in green, which also results in a miss. The system then writes the replaced block in the primary section to the backup section, and writes the backup block to L2 if it is dirty. The miss counter also increments.

A hit within either the primary or backup part causes the system to update the MRU state and counters. In Figure 7, block A (MRU[0]) is the most recently used, block B (MRU[1]) is the second most recently used, and so on. When the system accesses a block, the counter associated with the block's MRU state increments and the MRU state is updated.

For example, accessing 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 is the second most recently used. An access to C increments MRU[2] and changes the MRU state. The next access to C increments MRU[0] because it was just accessed, so it becomes the most recently used block while the access to D increments MRU[3].

At the end of an interval, a runtime routine reads the MRU counters and the miss counter, then uses this data 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 system chooses the best configuration for the next interval.

The MRU counters also permit calculation of the adaptive cache's actual performance loss compared to some fixed baseline. If the baseline is a subset of the adaptive cache, the system can use the MRU counters to determine what its performance would have been were it used. This value can be used to keep a running total of the adaptive cache's performance loss up to this point.

If the adaptive cache's performance loss is less than the target, the controller can be more aggressive in trading performance for energy. If its performance loss is excessive, the controller must reduce the overall loss by acting more conservatively. This accounting operation permits a tight bound on the performance loss while maximizing energy savings.

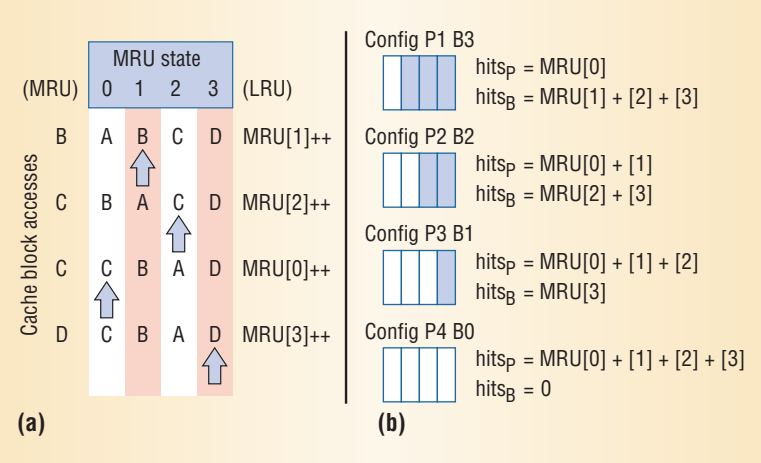


Figure 7. Updating MRU state and counters. (a) State changes and counter updates when accessing four different cache blocks. (b) Calculations performed to determine the number of primary and backup hits for each configuration. A configuration denoted as Px By, for example, has x primary and y backup partitions.

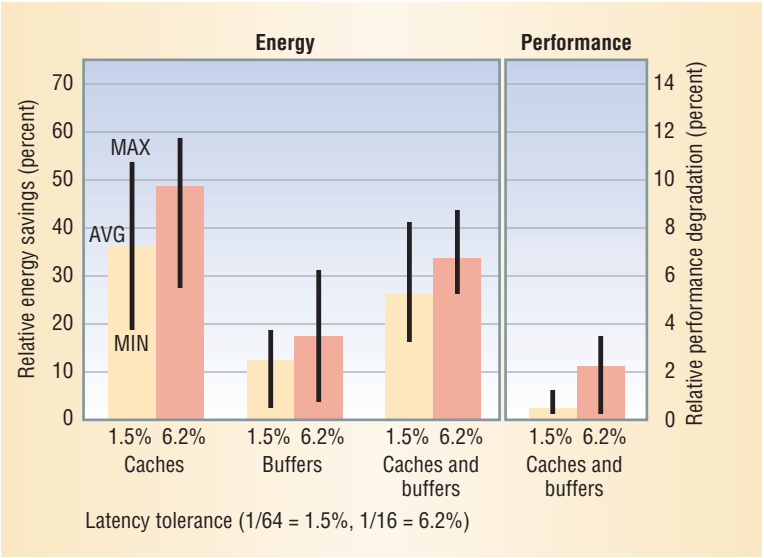


Figure 8. Adaptive structure energy savings and performance degradation across 14 benchmarks. Savings categories include caches only; queues, register files, and reorder buffers only; and both combined. The lines bisecting the bars show the range of values for the tested benchmarks.

The queues, register files, and reorder buffer use a variation of Ponomarev's feedback and control approach. When downsizing the register file, the system moves into an active partition the register values stored in the partition to be disabled. First, the system prevents any physical register in the partition to be disabled from being allocated to newly renamed instructions. Next, it executes a small runtime routine that performs the instruction *move rx, rx* for each logical register *rx*. This causes the system to read and transfer any logical register values

GRACE: A Cross-Layer Adaptation Framework for Saving Energy

Daniel Grobe Sachs, Wanghong Yuan, Christopher J. Hughes, Albert F. Harris III, Sarita V. Adve, Douglas L. Jones, Robin H. Kravets, and Klara Nahrstedt, University of Illinois at Urbana-Champaign

Adaptation techniques are commonly used for reducing energy in each layer of the system—hardware, network, operating system, and applications. Reaping the full benefits of a system with multiple adaptive layers requires a careful coordination of these adaptations. The Illinois GRACE project—Global Resource Adaptation through Cooperation—has developed a cross-layer adaptation framework to reduce energy while preserving desirable application quality for mobile devices running soft real-time multimedia applications.

A cross-layer adaptive system must balance two conflicting demands: adaptation scope and temporal granularity. For example, a global adaptation scope is desirable, but it can be expensive and so must be infrequent. Adapting infrequently, however, risks an inadequate response to intervening changes.

To balance this conflict, GRACE adopts a hierarchical approach, performing expensive global adaptations occasionally and inexpensive limited-scope adaptations constantly. This combination can achieve most of the benefits of frequent global

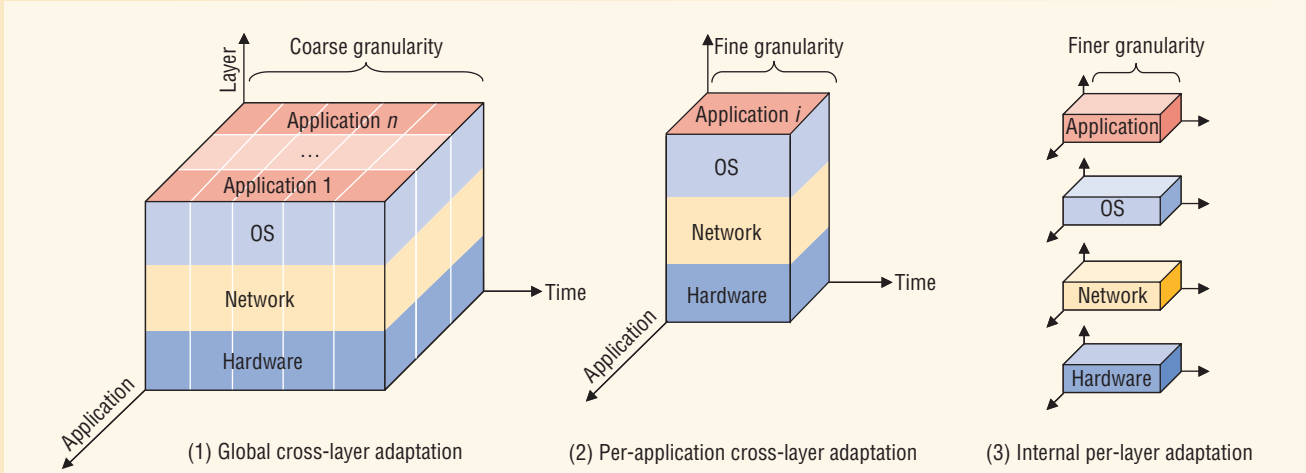


Figure A. Adaptation hierarchy. The framework's three layers exploit different adaptation scopes and temporal granularities.

stored in physical registers in the target partition to a newly allocated physical register from the enabled part of the register file.

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.

We plot these results as a function of the permissible target-performance degradation. The 1.5 percent and 6.2 percent values correspond to the power-of-two fractions 1/64 and 1/16, respectively. If power-of-two values are used for the performance degradation threshold, the hardware uses a shifter as the divide circuit. As expected, a higher target-performance degradation permits greater energy savings. Further, the ability of modern caches to hide latency with other work lets the actual performance degradation dip much lower than the permissible target.

The higher energy savings achieved in the caches stems from their greater overall energy compared to the buffers. For the 1.5 percent performance degradation target, the adaptive structures achieved a 28 percent energy savings, with only a 0.6 percent actual overall performance degradation. For the 6.2 percent target, the adaptive structures achieved a 34 percent energy savings with only a 2.1 percent performance loss.

To assess the cost and savings of adaptive processing, the energy savings must be determined for the processor as a whole. A modern superscalar processor's issue queues, reorder buffer, caches, and register files can easily consume more than 50 percent of the total chip power.[14] Using this figure as a conservative scaling factor for the energy results, an adaptive processing system can achieve an overall chip energy savings of roughly 14 percent in exchange for a 0.6 percent performance loss, or a 17 percent energy savings in exchange for a 2.1 percent performance loss.

All power-saving techniques can be compared to the common measure of a 3 to 1 power savings to performance degradation ratio that can be achieved simply by statically scaling the voltage. Assuming a linear relationship between frequency and voltage, this corresponds to a 2 to 1 energy savings to performance degradation ratio. Our sample adaptive processing system can achieve energy savings to performance degradation ratios from 8 to 1 up to

adaptation with lower overhead. As Figure A shows, GRACE supports three levels of adaptation, exploiting the natural frame boundaries in periodic real-time multimedia applications:

- *Global adaptation* considers all applications and system layers together but only occurs on large changes in system activity, such as application entry or exit.
- *Per-application adaptation* considers one application at a time and is invoked every frame, adapting all system layers to that application's current demands.
- *Internal adaptation* adapts only a single system layer, possibly considering several applications simultaneously, and is invoked several times per application frame—per network packet, for example.

All adaptation levels are tightly coupled, ensuring that the limited-scope adaptations respect the resource allocation decisions made through global coordination. Further, all adaptation levels use carefully defined interfaces so that no application or system layer needs to expose its internals to other parts of the system.

The initial GRACE-1 prototype[1] combines global application and CPU adaptations and internal scheduler and CPU adaptations, providing an increase in battery lifetime of 33 to 66 percent over previous systems. Separately, a combination of CPU and application adaptation at the per-application level[2] achieves energy savings of up to 20 percent and 72 percent, respectively, compared to either adaptation alone. Other experiments show the benefit of hierarchical adaptation within a single layer;[3] the addition of internal adaptation gave an energy reduction of up to 19 percent over a per-application CPU adaptation alone.

We are currently integrating these components along with an adaptive network layer to form a complete coordinated adaptive system.

Acknowledgment

This work is supported in part by the National Science Foundation under grant no. CCR-0205638.

References

1. W. Yuan et al., "Design and Evaluation of a Cross-Layer Adaptation Framework for Mobile Multimedia Systems," *Proc. Multimedia Communications and Networking*, SPIE—Int'l Society for Optical Engineering, 2003, pp. 1-13.
2. D.G. Sachs, S.V. Adve, and D.L. Jones, "Cross-Layer Adaptive Video Coding to Reduce Energy on General-Purpose Processors," *Proc. Int'l Conf. Image Processing*, Session WA-S2, IEEE Press, 2003, pp. 25-28.
3. R. Sasanka, C.J. Hughes, and S.V. Adve, "Joint Local and Global Hardware Adaptations for Energy," *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, ACM Press, 2002, pp. 144-155.

The authors are affiliated with the Computer Science and Electrical and Computer Engineering Departments at the University of Illinois at Urbana-Champaign. Contact them at grace@cs.uiuc.edu.

23 to 1, which indicates that adaptive processing's energy savings compare favorably with its performance cost. As discussed in the "GRACE: A Cross-Layer Adaptation Framework for Saving Energy" sidebar, combining these hardware adaptations with adaptations in other layers of the system provides further benefits.

To date, many adaptive processing techniques have focused exclusively on reducing dynamic power. In future process technologies, we expect that leakage power will rival dynamic power in magnitude—and adaptive techniques will be positioned to address both. For example, an adaptive processing system can apply voltage-gating directly to the issue queues, reorder buffer, and register files because the system ensures disabled partitions are empty before turning them off. This technique can also be applied to the L1 I-cache, while an approach such as drowsy caches[6] can be used in the L1 D-cache and L2 cache to preserve their state.

Adaptive processors require a modest amount of additional transistors. Further, because adaptation occurs only in response to infrequent trigger events, the decision logic can be placed into a low-leakage state until a trigger event occurs. These characteristics make adaptive processing a promising approach for saving both dynamic and leakage energy in future CMOS technologies. ■

References

1. D.H. Albonesi, "Dynamic IPC/Clock Rate Optimization," *Proc. 25th Int'l Symp. Computer Architecture*, IEEE CS Press, 1998, pp. 282-292.
2. D.H. Albonesi, "The Inherent Energy Efficiency of Complexity-Adaptive Processors," *Proc. 1998 Power-Driven Microarchitecture Workshop*, 1998, pp. 107-112.
3. B. Xu and D.H. Albonesi, "A Methodology for the Analysis of Dynamic Application Parallelism and Its Application to Reconfigurable Computing," *Proc. SPIE Int'l Symp. Reconfigurable Technology: FPGAs for Computing and Applications*, SPIE Press, 1999, pp. 78-86.
4. A. Buyuktosunoglu et al., "A Circuit-Level Implementation of an Adaptive-Issue Queue for Power-Aware Microprocessors," *Proc. 11th Great Lakes Symp. VLSI*, ACM Press, 2001, pp. 73-78.
5. M.D. Powell et al., "Reducing Leakage in a High-

Performance Deep-Submicron Instruction Cache," *IEEE Trans. VLSI Systems*, vol. 9, no. 1, 2001, pp. 77-89.
6. N.S. Kim et al., "Drowsy Instruction Caches—Leakage Power Reduction Using Dynamic Voltage Scaling and Cache Sub-Bank Prediction," *Proc. Int'l Symp. Microarchitecture*, IEEE CS Press, 2002, pp. 219-230.
7. D. Ponomarev et al., "Reducing Power Requirements of Instruction Scheduling through Dynamic Allocation of Datapath Resources for Low Power," *Proc. Int'l Symp. Microarchitecture*, IEEE CS Press, 2001, pp. 90-101.
8. D. Folegnani and A. Gonzalez, "Energy-Effective Issue Logic," *Proc. Int'l Symp. Computer Architecture*, IEEE CS Press, 2001, pp. 230-239.
9. M.C. Huang, J. Renau, and J. Torrellas, "Positional Adaptation of Processors: Application to Energy Reduction," *Proc. Int'l Symp. Computer Architecture*, IEEE CS Press, 2003, pp. 157-168.
10. R. Balasubramonian et al., "Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures," *Proc. 33rd Int'l Symp. Microarchitecture*, IEEE CS Press, 2000, pp. 245-257.
11. A.S. Dhodapkar and J.E. Smith, "Managing Multiconfiguration Hardware via Dynamic Working Set Analysis," *Proc. Int'l Symp. Computer Architecture*, IEEE CS Press, 2002, pp. 233-244.
12. R. Sasanka, C.J. Hughes, and S.V. Adve, "Joint Local and Global Hardware Adaptations for Energy," *Proc. Int'l Conf. Architecture Support for Programming Languages and Operating Systems*, ACM Press, 2002, pp. 144-155.
13. S. Dropsho et al., "Integrating Adaptive On-Chip Storage Structures for Reduced Dynamic Power," *Proc. 11th Int'l Conf. Parallel Architectures and Compilation Techniques*, IEEE CS Press, 2002, pp. 141-152.
14. P. Bose et al., "Early-Stage Definition of LPX: A Low-Power Issue-Execute Processor," *Proc. Workshop Power-Aware Computer Systems*, Springer, 2002, pp. 1-17.

David H. Albonesi *is an associate professor in the Department of Electrical and Computer Engineering at the University of Rochester. Contact him at albonesi@ece.rochester.edu.*

Rajeev Balasubramonian *is an assistant professor in the School of Computing at the University of Utah. Balasubramonian received a PhD from the University of Rochester. Contact him at rajeev@cs.utah.edu.*

Steven G. Dropsho *is a postdoctoral researcher in the Department of Computer Science at the University of Rochester. Contact him at dropsho@cs.rochester.edu.*

Sandhya Dwarkadas *is an associate professor in the Department of Computer Science at the University of Rochester. Contact her at sandhya@cs.rochester.edu.*

Eby G. Friedman *is a distinguished professor in the Department of Electrical and Computer Engineering at the University of Rochester. Contact him at friedman@ece.rochester.edu.*

Michael C. Huang *is an assistant professor in the Department of Electrical and Computer Engineering at the University of Rochester. Contact him at michael.huang@ece.rochester.edu.*

Volkan Kursun *is a PhD candidate in the Department of Electrical and Computer Engineering at the University of Rochester. Contact him at kursun@ece.rochester.edu.*

Grigorios Magklis *is a researcher at the Intel Barcelona Research Center. Magklis received a PhD from the University of Rochester. Contact him at grigoriosx.magklis@intel.com.*

Michael L. Scott *is a professor in the Department of Computer Science at the University of Rochester. Contact him at scott@cs.rochester.edu.*

Greg Semeraro *is an assistant professor in the Department of Computer Engineering at the Rochester Institute of Technology. Semeraro received a PhD from the University of Rochester. Contact him at gpseec@ce.rit.edu.*

Pradip Bose *is a research staff member at the IBM T. J. Watson Research Center. Contact him at bose@us.ibm.com.*

Alper Buyuktosunoglu *is a research staff member at the IBM T.J. Watson Research Center. Contact him at alperb@us.ibm.com.*

Peter W. Cook *is a manager at the IBM T. J. Watson Research Center. Contact him at pwcook@us.ibm.com.*

Stanley E. Schuster *is a research staff member at the IBM T.J. Watson Research Center. Contact him at schustr@us.ibm.com.*