Runtime Reconfiguration Techniques for Efficient General Purpose Computation

Bingxiong Xu and David H. Albonesi
Dept. of Electrical and Computer Engineering
Computer Studies Building, Box 270231
University of Rochester
Rochester, NY 14627-0231
Phone: (716) 275-3870
{bixu,albonesi}@ece.rochester.edu

Abstract: Because of their widespread use, general purpose microprocessors are called upon to execute an increasingly diverse set of applications. Due to their static organization, these devices often operate very inefficiently under these conditions, which results in lost performance opportunities and extra energy dissipation. By exploiting the partitioning of major microprocessor hardware structures, and applying runtime reconfiguration techniques, microprocessor efficiency can be greatly improved while retaining the benefits of fast clock speed, dense functionality, and mature software and chip development tools. While the functionality of each hardware structure remains fixed, its *complexity* is configured to match application requirements at runtime. Two applications of this approach demonstrate significant improvements in both energy efficiency and performance.

Introduction

The performance of general purpose microprocessors continues to increase at a rapid pace. In the last 15 years, the performance of these devices has increased at a rate of roughly 1.6 times per year [11]. These performance gains have been fueled by continuing advances in semiconductor technology, which allow for additional on-chip functionality and faster clock speeds with each new microprocessor generation. As significant as these technological advances is the massive amount of design experience and infrastructure accumulated in developing complex out-of-order speculative processors and associated software and development tools. This mature machinery currently allows new generations of microprocessor hardware and supporting compiler and operating system software to be delivered every 3-4 years.

This success story has led to the commoditization of the microprocessor and its proliferation throughout society. A single core processor design may find its way into such myriad environments as desktop PCs, web servers, supercomputers, portable devices, and printers. Thus, general purpose microprocessors are called upon to run an increasingly diverse workload, from traditional business and scientific programs to data mining, multimedia, virtual work environments, and collaborative computing applications. These applications may vary markedly in their characteristics, including their instruction-level parallelism and memory access patterns. For a given chip area, the hardware features that provide the best balance between exploiting these characteristics and optimizing clock speed and energy efficiency is a function of what application, or portion of an application, is running at any given moment.

However, because a conventional microprocessor is designed with unchanging, *static* hardware structures, the task of designing such a device is one of compromises and tradeoffs in an attempt to produce a design that provides *best overall* performance across a diverse workload. As measured by the proliferation of the general purpose microprocessor, and its annual 1.6 times performance improvement, it can be argued that this “best overall” approach has thus far been successful. However, the cost of this compromise approach is the inefficient operation of microprocessor hardware structures, a design issue that is growing in importance with each processor generation.

A processor hardware structure, such as a branch prediction table, cache, or TLB, operates inefficiently when its elements are not well utilized by a given task, and when removing a significant fraction of the resource has little impact on the number of cycles required to complete the task. Consider the core processor hardware for instruction issue, execution, and commitment. The utilization of these resources may depend on many factors, including the instruction and data supply demands and the inherent data dependencies of the application. Simulation-based studies [1, 18] and measurements of real microprocessors [4, 8] reveal that the utilization of these core resources, as well as other resources such as caches, vary widely from application to application. However, in order to more fully examine processor efficiency, some insight also needs to be acquired on how effectively these resources exploit parallelism during different periods of individual application execution.

Evaluating Processor Efficiency Via Dynamic Parallelism Analysis

By quantifying the *dynamic parallelism*, the parallelism of an application during different periods of execution, insight can be gained as to how processor efficiency changes during the execution of a single program. To perform this analysis, parallelism measurements are periodically taken during application simulation. A plot of these measurements as a function of the number of executed instructions shows how parallelism changes during execution. A comparison of parallelism plots for different hardware configurations reveals the relative

Table 1: Simulator hardware parameters for *full config*.

Parameter	Value
fetch width	1024
decode width	1024
RUU size	1024
issue width	1024
integer ALUs	1024
integer multiplier/dividers	512
flt. pt. ALUs	1024
flt. pt. mult/div	512
memory ports	512
commit width	1024

effectiveness of these configurations in exploiting parallelism at various phases of application execution.

Three aggressive processor designs, reflecting the total processor resources that might be available in the future, are modelled using a detailed, execution-driven simulator [7] of a speculative out-of-order processor that implements a superset of the MIPS instruction set. The simulator uses a centralized instruction queue called the Register Update Unit (RUU) from which decoded instructions are issued and in which results are held before they are committed. The RUU is similar to the Instruction Reorder Buffer used in the HP PA-8000 dynamic superscalar microprocessor [14].

The three processor designs vary in their support for instruction issue, execution, and commitment. The model *full config* uses the parameters in Table 1, while the models *half config* and *quarter config* have one-half and one-quarter, respectively, of the hardware resources of *full config*. The instruction fetch, decode, branch predict, and data fetch mechanisms of these configurations are only limited by the width of the processor instruction and data paths; otherwise, perfect instruction and data streams are provided to the core processor hardware. In addition, all instructions execute in a single cycle. Thus, outside of data dependencies, there are few inhibitors to exploiting application parallelism.

Despite this fact, the snapshots in Figure 1 of the dynamic parallelism of six of the SPEC95 benchmarks, demonstrate both the uniform and non-uniform effects of various levels of hardware support for extracting parallelism. For *tomcatv* and *fpppp*, the most aggressive model has a large and uniformly beneficial impact on parallelism. For *turb3d*, in which the overall parallelism of the full configuration is less than 4% greater than that with half the resources, there is little difference between these two configurations throughout execution. Even the *quarter config* model provides similar performance to the more aggressive models during some periods of execution. For *li*, *ijpeg*, and *hydro2d*, the relative effectiveness of the three hardware configurations in exploiting parallelism varies non-uniformly during execution. For *li* for example, the full configuration achieves a parallelism that is over 22% greater than that with one-quarter the hardware resources during the 400 million instruction execution period that we studied, and 35% greater once the first 181.5M instructions have executed. Yet, there are regular periods of execution that are significant in length during which there is no discernible difference in parallelism between these configurations. Similar non-uniformity is observed with *ijpeg* and *hydro2d*.

These results demonstrate how the efficiency of the core processor resources for extracting application parallelism can vary from application to application, and even within the execution of an individual application. For *tomcatv* and *fpppp*, the 30-50% parallelism boost provided by the *full config* model over the *half config* model justifies the use of the more aggressive configuration for these applications. However, for *turb3d*, the *full config* model provides little benefit over the simpler models when one considers the additional hardware cost of the more complex model. For *li*, *ijpeg*, and *turb3d*, the use of the most aggressive hardware is beneficial during some periods of execution, yet for others, the lesser models

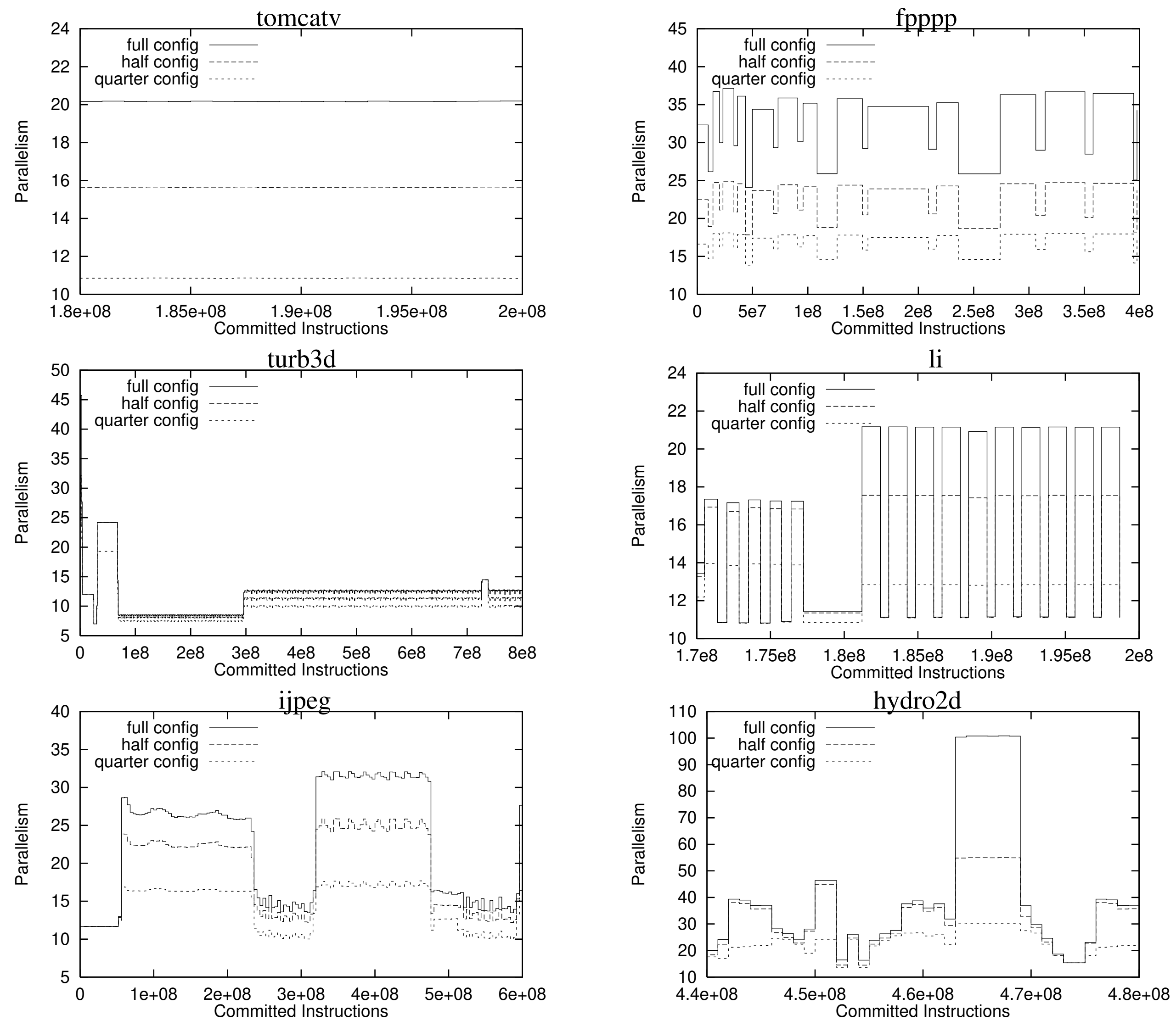


Figure 1: The dynamic parallelism of six different SPEC95 benchmarks under three hardware configurations.

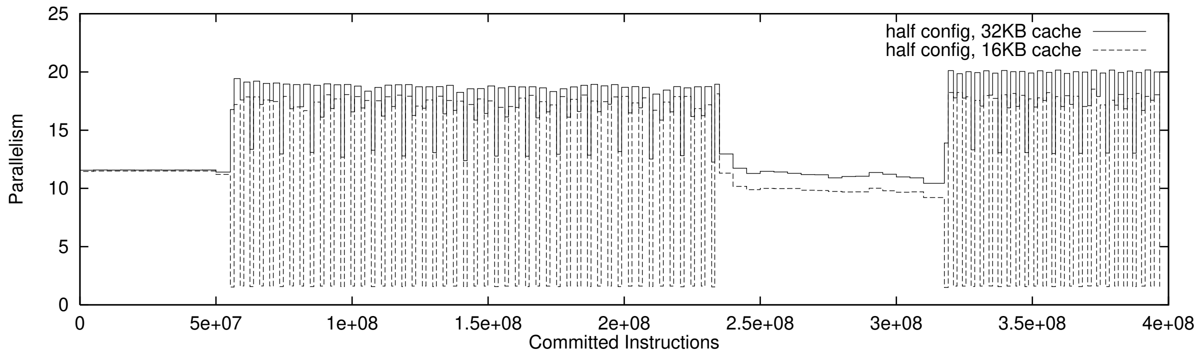


Figure 2: The dynamic parallelism of *ijpeg* with 16KB and 32KB L1 Dcaches.

provide almost identical performance.

Cache efficiency can vary widely during application execution as well. Figure 2 shows the dynamic parallelism of a portion of *ijpeg* for two models: *half config* with a direct-mapped 16KB L1 data cache, and *half config* with a 32KB L1 data cache. There are three execution phases, representing different phases of the compression/decompression process, during which the difference in parallelism between these two caches varies widely. During the startup phase of the application, which lasts for the first 55 million instructions, there is less than a 1% different in performance between the two organizations. During the second phase, which lasts until about 230 million instructions have executed, two periods alternate in sequence. These periods are about two and three million instructions in length, and cache effectiveness differs radically between them. For the shorter periods, the 32KB cache outperforms the 16KB cache by about a factor of eight. However, during the longer periods, performance only differs by about 5%. During the next major phase, there is a steady 15% difference between the organizations. The second phase then repeats.

The results in this section illustrate how for some applications, a particular hardware resource may significantly improve performance throughout execution. For other applications, performance may not improve at all, or the improvement may be sporadic during execution.

Although the incorporation of the resource may be justified based on overall performance gain, for any given application the resource may operate very inefficiently. Thus, despite the impressive advances made by conventional microprocessors, their static nature renders them far from ideal in terms of delivering the best possible performance in the most efficient manner.

The Growing Importance of Processor Efficiency

Achieving efficient microprocessor operation is important for two reasons. First, there is a fundamental performance tradeoff between achieving a high instructions per cycle (IPC) rating with complex hardware versus a high clock rate with simpler hardware. A design that is overly complex for a given task fails to reach its full speed potential due to this extra complexity. A design that fails to exploit the parallelism or data locality of a program, when the level of integration is such that this can be achieved, may expend additional cycles than are necessary to execute the program. A modern microprocessor finds the middle ground of this tradeoff where most applications of interest perform well, but the inevitable result is that the tradeoff is far from optimal for many of these applications. One can almost always conjecture on an alternative, better performing design for any given application.

Second, the energy dissipation of a microprocessor is proportional to the switching capacitance of the hardware, which is a function of the amount of hardware functionality and the number of signal transitions that take place within it. A hardware function that is overdesigned for a given task wastes energy activating more logic and memory cells than are necessary to perform the task, while a hardware function that is too simple wastes energy performing an exceedingly large number of operations. Current microprocessors use conditional clocking in which hardware functions are partitioned into smaller structures, a subset of which are selectively enabled each cycle via hardware control. However, conditional clocking creates large and rapid fluctuations in power supply current which significantly complicates the power delivery system design due to transient noise issues [10, 17]. The Alpha 21264 microprocessor required voltage and ground planes and the incorporation of decoupling capacitors that comprised 15-20% of the total die area due to the transient effects of conditional clocking. Embedded microprocessors are also subject to these transient effects, which may eventually require expensive solutions that increase their cost and/or limit portability.

As semiconductor technology has continued to improve, allowing for increased integration and higher clock rates, making efficient use of processor resources has become more critical than ever. Recent design decisions, such as the split instruction queues in the Mips R10000 and Alpha 21264 and the dual integer clusters in the 21264, provide evidence of how architects are struggling to reconcile increased functionality and increased clock rates in current designs. This is largely due to the limitations in scaling wire delays with new process generations, which will become even more of a challenge in the future [15]. Furthermore, chip functionality and clock rates have increased to the point where unless countermeasures are taken, power dissipation may eventually limit the functionality that may be included on a microprocessor [10, 17]. By intelligently incorporating configurability into a conventional microprocessor, processor efficiency can be significantly improved while retaining the dense functionality and high clock rate that are critical for good general purpose computing performance.

Incorporating Configurability Into Commodity Microprocessors

In contrast to the static organization of conventional processors, Configurable Computing Machines (CCMs) consist of flexible hardware whose functionality can be dynamically adapted to changing program characteristics. A typical CCM contains a set of configurable functional blocks and a configurable interconnect. Many of these machines use FPGAs in order to finely tune the hardware functionality on-the-fly to match algorithms from applications such as signal processing and imaging. However, the low density, slow clock speed, and long reconfiguration times of most CCMs make them incompatible with many general purpose applications, which benefit more from the high density and high clock rate of static conventional microprocessors than from the flexibility afforded by CCMs. In addition, the development of many CCMs requires different algorithms, programming models, and/or development tools than those that have been successfully used to produce many generations of commodity microprocessors.

However, techniques from configurable computing can be effectively applied to conventional microprocessors in order to provide for better application-specific tailoring of the *hardware complexity.* Specifically, techniques for runtime reconfiguration (RTR) can be used to dynamically adapt the complexity of the processor hardware to match changing application characteristics, such as those evident in Figure 1. Thus, design techniques for RTR and conventional microprocessors can be combined so as to:

- Produce clock rates that are competitive with conventional microprocessors;
- Implement aggressive hardware support for exploiting parallelism and memory locality;
- Dynamically configure the hardware complexity and (optionally) the operational speed of the chip at runtime;
- Limit reconfiguration overhead to tens of clock cycles in order to exploit the rapid changes in hardware requirements demonstrated in Figure 1; and
- Use the mature programming models and design methodologies of conventional micro-

processors.

The above criteria imply starting with a conventional microprocessor design and incorporating a limited level of adaptability in order to overcome the aforementioned limitations of conventional static devices. This philosophy, which lies at the coarse-grain end of the CCM spectrum, we refer to as *Complexity-Adaptive Processing* or *CAP* because its central theme is allowing for hardware complexity to be dynamically adapted at runtime to match application requirements. In contrast to conventional microprocessor design, CAP provides aggressive hardware support for exploiting parallelism and memory locality in an *on-demand* fashion, running with a subset of this full support enabled during lower demand periods. With this *performance on-demand* approach to microprocessor design, a significant reduction in the switching energy can be realized by placing non-critical functionality into a quiescient state during appropriate periods of execution. Furthermore, if a *dynamic* clocking circuit whose frequency can be rapidly switched is incorporated into the design, then a faster clock can be enabled on-the-fly during low demand periods. This ability to dynamically trade off hardware complexity and clock speed at runtime can significantly improve microprocessor performance.

Another facet of CAP techniques is the exploitation of the properties of a modern microprocessor in order to limit configuration overhead. These properties are discussed in the next section.

Exploiting Microprocessor Partitioning for High-Speed Adaptivity

In order to adapt quickly enough to meet the changing requirements of general purpose applications, and to retain the density and clock speed of conventional microprocessors, Complexity-Adaptive Processing exploits high-speed microprocessor circuit design techniques

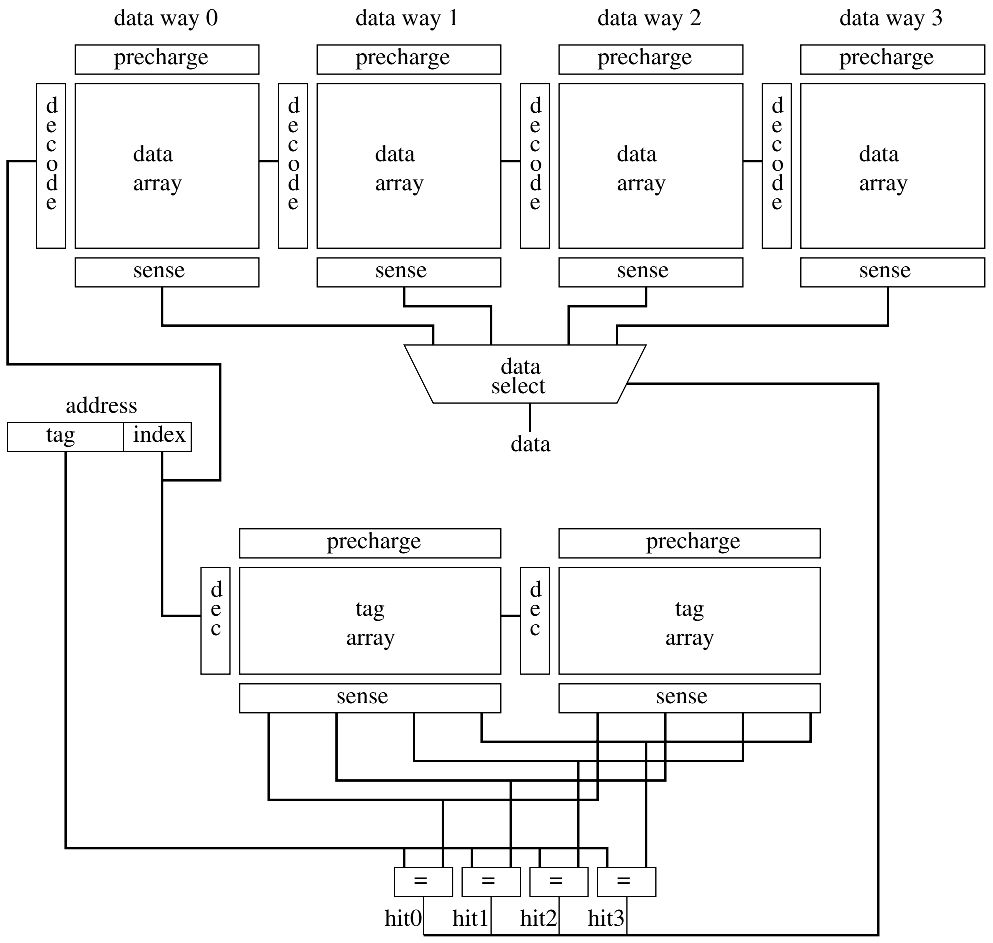


Figure 3: A four-way set associative cache with the data array vertically partitioned and the tag array horizontally partitioned.

to provide configurability at low cost. The functionality of a modern microprocessor is dominated by large RAM and CAM-based structures such as caches, TLBs, branch predictor tables, register rename tables, instruction queues, and register files. Each of these memory structures is often partitioned into multiple memories called *subarrays* for speed purposes. Such partitioning decreases the long wordline and/or bitline delays of a single large RAM, the net result of which is a faster access time. For example, Figure 3 shows a four-way set associative cache with subarray partitioning. The data array is vertically sliced into four subarrays, each with its own local wordline decoder and one-quarter the sense amplifiers

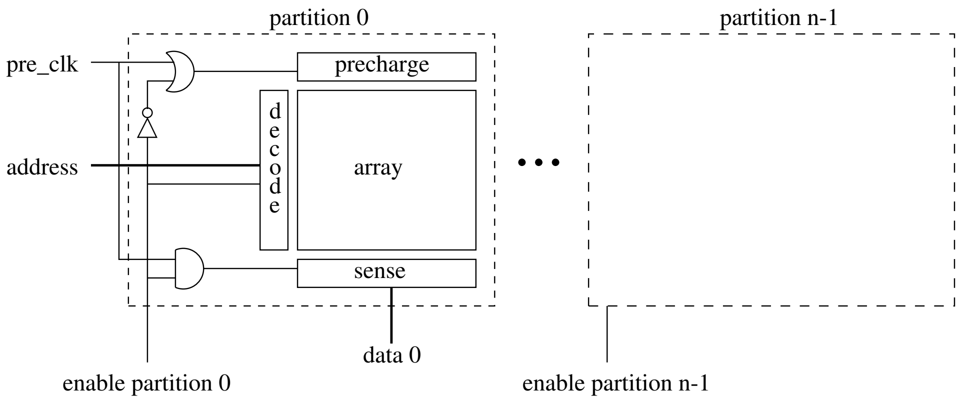


Figure 4: A memory structure organized as n partitions, each of which can be individually enabled or disabled.

of a single data array. Here, each wordline is roughly one-quarter the length of that in a single array (although there may be a global wordline that spans one-half to the full array length). Two tag subarrays are formed by segmenting the bitlines, resulting in a halving of the decoder width but a doubling of the number of sense amps relative to a single tag array. Only one of the two sets of tag sense amplifiers associated with the same column is activated during each access.

These properties of major microprocessor hardware structures can be exploited in order to provide low-cost configurability. In particular, a small amount of gating logic can be used to rapidly enable or disable subarray partitions during execution. This logic transforms a conventional static structure into a *dynamic* one whose complexity can be changed on-the-fly at runtime. In a conventional CCM, the operation performed by each function block and the interconnect are configurable. In contrast, our goal is for the function and interconnect to be fixed as in a conventional microprocessor but for its *complexity* to be configurable.

Figure 4 shows a dynamic memory structure which has been partitioned into subarrays for speed purposes, and the small amount of additional circuitry needed to selectively disable each memory partition, which is comprised of one or more subarrays. The gating logic in this diagram is based on that used to conditionally clock data subarrays in the Alpha 21164 L2 cache [5]. Each *enable partition* signal controls the enabling of one of the n memory partitions. If a particular *enable partition* line is at logic zero, then that partition is not precharged, no word lines are selected, and its sense amps are prevented from firing. Thus, no switching activity ensues and thus this partition dissipates essentially no dynamic power.

Note that regularity in microprocessor logic structures and buses can be similarly exploited. For example, a large logic function, such as a wide priority encoder, may be designed as a tree of smaller structures. As with memory structures, this function can be made dynamic by adding *enable partition* signals to different groups of these smaller structures. By exploiting this natural partitioning of high-speed circuits, only very minimal changes are required to achieve configurability. However, additional hardware and software mechanisms are required to adapt these dynamic structures at runtime to match changing application requirements.

Elements of Complexity Adaptive Processing

A conventional approach to processor design would implement decoder logic at each pipeline stage to control dynamic structures on a cycle-by-cycle basis. The analysis of the benchmarks of Figure 1 as well as other SPEC95 benchmarks reveals that major changes in parallelism and cache requirements frequently occur in applications at a medium-grain level, no faster than every 100,000 instructions [20], or every 4000 clock cycles on a future processor that sustains an IPC of 25. At this rate, to limit the reconfiguration overhead to 1% of the execution time, reconfiguration can take as much as 40 clock cycles. With the exception of some functions, such as data caches and instruction queues which may require time to preserve data before partitions are disabled, partitions can be enabled or disabled in only a few cycles using the mechanisms of Figure 4.

For these reasons, Complexity-Adaptive Processing uses the RTR approach of software control of the configurable hardware layer, obviating the need for hardware decision logic. The simple mechanism used in the hardware to enable or disable partitions leaves ample time for executing special instructions to load the CR while maintaining low configuration overhead. Thus, a CAP implementation is a combined hardware/software system that is composed of most, if not all, of the elements shown in Figure 5:

- Dynamic hardware structures;
- Conventional static hardware structures;
- Performance counters which track the performance of each dynamic structure as well as the overall processor, and which are readable via special instructions and accessible to the control hardware;
- A Configuration Register (CR) which is readable and writable via special instructions and by the hardware, and whose outputs drive the *enable partition* inputs for each dynamic structure and (optionally) control the clock speed of the chip;
- An optional dynamic clocking system whose frequency is controlled via particular CR bits; a change in these bits causes a sequence in which the current clock is disabled and the new one started after an appropriate settling period;
- An instruction set consisting of conventional instructions augmented with special instructions for loading the CR and reading the performance counters;
- Configuration control, implemented in the compiler, the runtime system, and (optionally) the dynamic reconfiguration control logic (DRCL), that acquires information about the application and uses predetermined knowledge about the complexity and clock speed of each hardware configuration to create a *configuration schedule* that

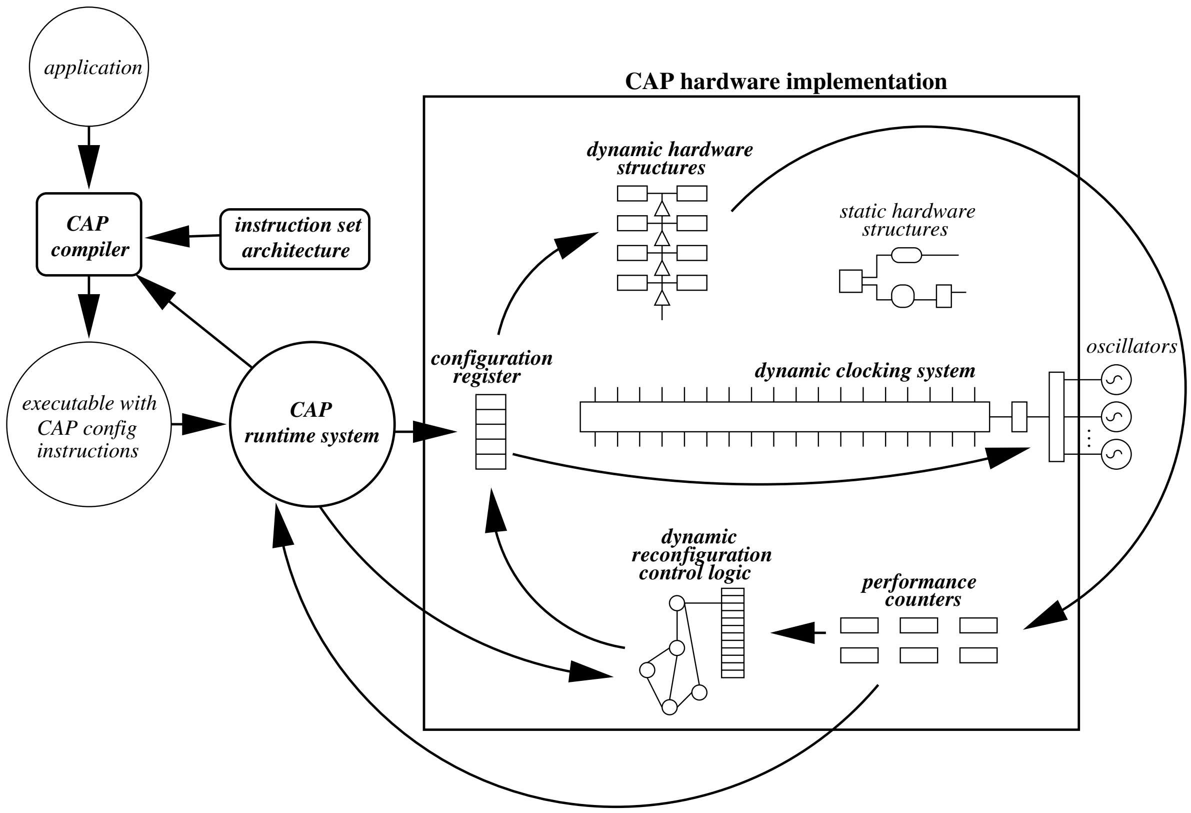

application
CAP compiler
instruction set architecture
executable with CAP config instructions
CAP runtime system
CAP hardware implementation
dynamic hardware structures
static hardware structures
configuration register
dynamic clocking system
oscillators
dynamic reconfiguration control logic
performance counters

Figure 5: Overall elements of a CAP hardware/software system.

matches the hardware complexity to application requirements dynamically during its execution.

The process of compiling and running an application on a CAP machine is as follows. The CAP compiler analyzes application hardware requirements for different phases of its execution. For example, it may analyze dynamic data cache requirements based on working set analysis, or determine the dynamic parallelism based on the data and control flow graphs. With this information, and knowledge about available hardware configurations, the compiler determines whether it can with good confidence create an effective configuration schedule, specifying at what points within the application the hardware should be reconfigured, and to which organizations. The schedule is created by inserting special instructions at particular points within the application that load the CR with the desired configuration. In cases where dynamic runtime information is necessary to determine the schedule, this task is performed by the runtime system or (optionally) the DRCL. For example, TLB configuration scheduling may be best handled in conjunction with the TLB miss handler, based on runtime TLB performance monitoring, while the optimal branch predictor size may in some cases be best determined by a hardware-based predictor. Although software control is preferred due to its low hardware overhead, CAP implementations may choose to use DRCL in order to rapidly predict the optimal configuration based on runtime information.

Through these various mechanisms, the CR is loaded at various points during application execution, resulting in configuration of dynamic structures and changes in clock frequency when a dynamic clocking system is implemented. For runtime control, the performance counters are queried at regular intervals of operation, and using history information about past decisions, a prediction is made about the configuration that will perform best over the next interval. The runtime system may also use a binary rewriting tool to perform continuous optimization of the application.

In contrast to conditional clocking, the enabling of the partitions in a structure such as that in Figure 4 is under software control, and changes in enabled and disabled functionality occur over the longer time intervals in which application requirements change (typically hundreds of thousands of instructions or more). Thus, the transient current effects encountered with conditional clocking can be controlled by gradually transitioning to the new configuration. The transition from a configuration in which the minimal number of partitions are enabled to one in which all are enabled can be made in several intermediate steps, thereby lengthening the period over which this significant change in supply current is made. Because each reconfiguration requires a few tens of cycles, and changes occur over these long execution periods, this extra cost can be absorbed with negligible performance impact. Conditional clocking may also result in a delay penalty in cases where parallel access of partition groups is desired, *e.g.*, in attempting to select only the data way in which there is a hit in a set associative cache. With the CAP approach, in contrast, a subset of the data cache ways can be enabled for a given period of execution while still allowing for parallel tag-data array access. This is discussed in the next section in which some specific applications of Complexity-Adaptive Processing are presented.

Applications of Complexity-Adaptive Processing

Complexity-Adaptive Processing can be applied to isolated hardware structures or to large slices of a microarchitecture, to synchronous, asynchronous, or mixed timing systems. These techniques can be used at the process level or within an individual application, to improve performance, energy efficiency, or both. Two CAP applications are described in this section. The first addresses improving the energy efficiency of the L1 data cache in a present-day microprocessor on an application-by-application basis. The second explores how the core processor resources of a future microprocessor can be partitioned to provide more optimal complexity and speed balance dynamically at runtime. Additional applications of Complexity-Adaptive Processing can be found in [1].

Selective Cache Ways

As was shown in Figure 2, the L1 cache requirements of a particular application may vary widely during execution, with periods of high demand followed by periods in which performance will not suffer considerably when only a subset of the full cache is enabled. Cache requirements may also vary considerably from application to application [1]. During low demand periods, considerable energy may be wasted accessing more cache cells than are needed by the application. This inefficient operation may occur even with conditional clocking, which, in addition, increases transient current noise as discussed in the previous section. Accessing the tag array first, and then activating only the data partition containing the requested data, considerably reduces energy but at the cost of higher cache latency. The result can be a significant performance degradation for many applications.

The energy efficiency of a conventional L1 cache can be dramatically improved with only a small performance impact and without the high transient currents of conditional clocking by disabling cache subarray partitions during appropriate periods of execution. One approach is to allocate the individual ways of a set associative cache in an on-demand fashion according to application requirements. This CAP technique, called *Selective Cache Ways* [3], allows for full-speed cache operation as the tag and data arrays are accessed in parallel as in a conventional cache. Thus, by anticipating low demand periods, the performance degradation of this approach can be kept to a tolerable level.

Hardware Organization

Figure 6 is an overall diagram of a four-way set associative cache using selective cache ways. The wordlines of the data array are segmented four times according to the optimal partitioning determined by the Cacti cache cycle time model [19], creating four separate data way elements. The bitlines of each data way may be segmented as well, although this is not shown in the diagram. Note however, that the tag portion of the cache (which also includes the status bits) is identical to that of a conventional cache. Cacti-based timing estimates indicate that for the cache organizations studied, segmenting the tag wordlines will result in a significant cache cycle time degradation relative to the optimal tag partitioning. For these reasons, this CAP approach only saves energy in the data portion of the cache, but this comprises roughly 90% of the total energy dissipation for the cache organizations that were studied.

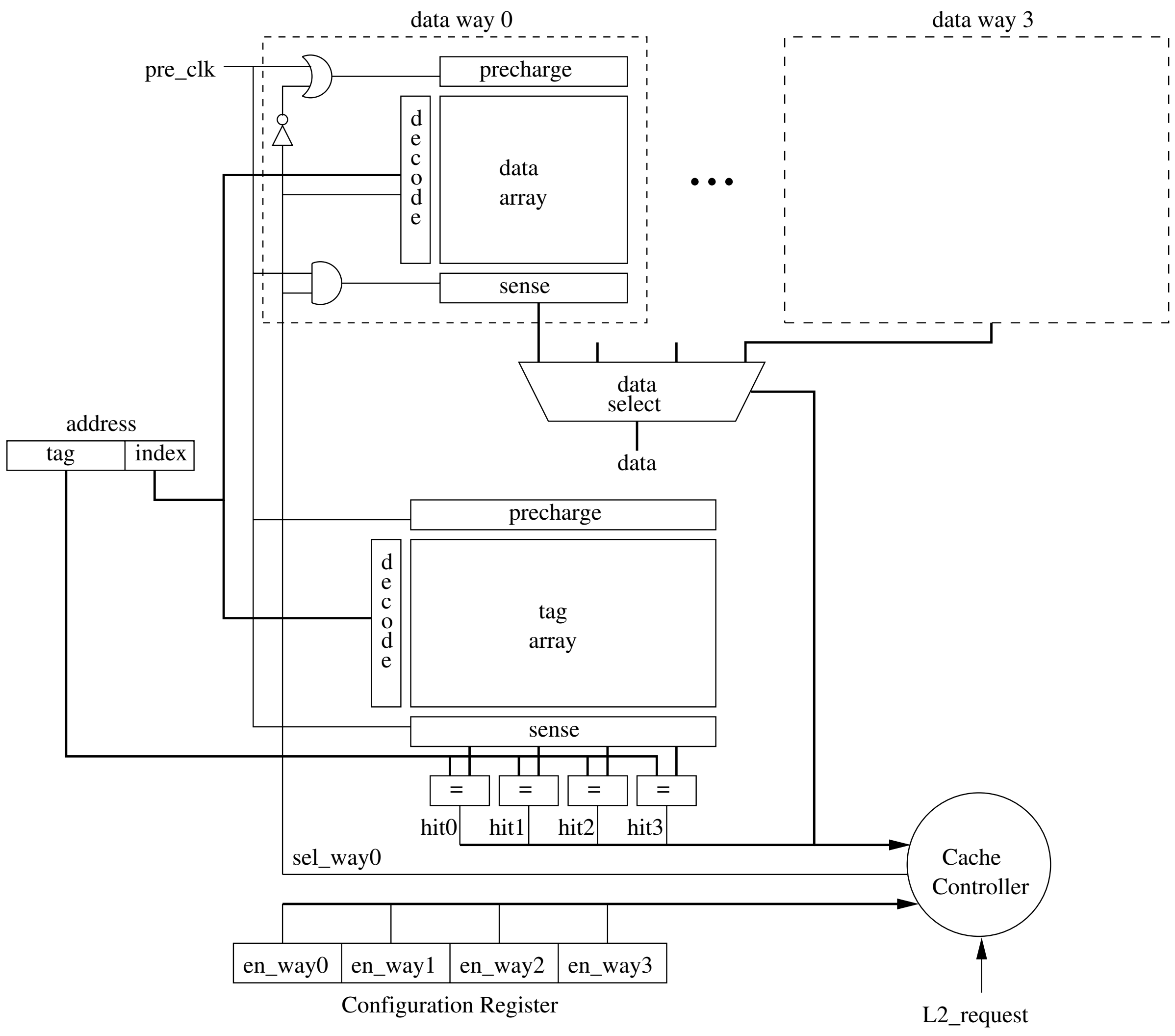


Figure 6: A four-way set associative cache using selective cache ways. The details for data ways 1-3 are identical to way 0 but are not shown for simplicity.

The Configuration Register contains four bits in this example (*en_way0*, *en_way1*, *en_way2*, and *en_way3*), each of which signals the Cache Controller to enable a particular way, and thereby allow it to operate. If a particular way enable bit is set to zero, the *sel_way* signal for that way is also zero (except in circumstances related to data sharing and coherency; mechanisms for handling these situations are discussed in detail in [3]). Therefore, no data is selected from a disabled way and its data array dissipates no dynamic power. The replacement decision logic within the Cache Controller also ensures that no new data is allocated for a disabled way.

Saving Energy With Selective Cache Ways

Cache energy is reduced with selective cache ways during periods where the energy savings of disabling cache ways outweighs the increase in energy due to servicing additional cache misses, and where some small performance degradation can be tolerated.

The benefits of selective cache ways is assessed by simulating a four-way out-of-order speculative processor with a two-level cache hierarchy that roughly corresponds to a current high-end microprocessor such as the HP PA-8000 [14] and Alpha 21264 [13]. Table 2 shows the simulator parameters for the memory hierarchy. Selective cache ways is implemented for only the L1 Dcache. The data array of the L2 cache is implemented as 16 partions, only one of which is selected for each access, similar to the approach used in the Alpha 21164 on-chip L2 cache [5].

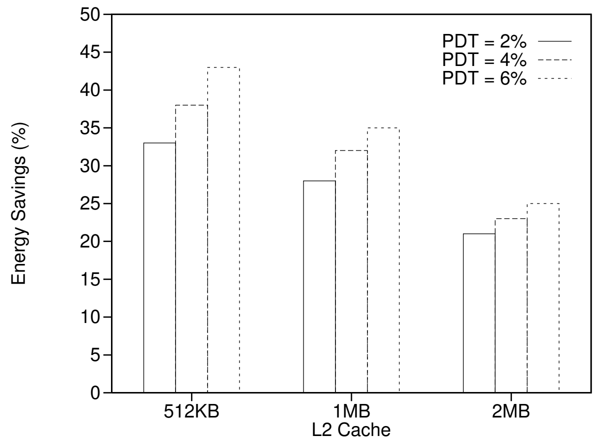
Cache energy dissipations are calculated using a detailed cache energy dissipation model [12] that uses technology and layout parameters as well as counts of various cache events (hits, writebacks, *etc.*) as inputs. These event counts, in addition to performance results, are gathered from simulations of eight benchmarks: the SPEC95 benchmarks *compress*, *ijpeg*, *li*,

Table 2: Simulated memory hierarchy parameters.

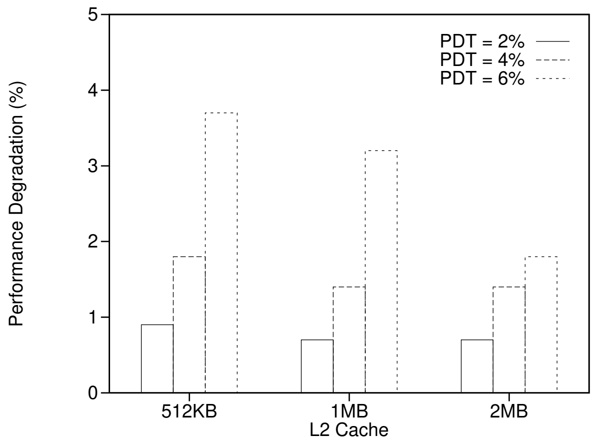
Cache	Organization
L1 Icache	64KB, 4-way set assoc, 32B block, random, 1 cycle latency
L1 Dcache	64KB, 4-way set assoc, selective cache ways, 2 ports, 32B block, random, 1 cycle latency
L2 cache	512KB, 1MB, or 2MB, 4-way set assoc, 32B block, LRU, 15 cycle latency, 16 partitions
main memory	16B bus width, 75 cycle initial latency, 2 cycles thereafter

turb3d, *mgrid*, *fpppp*, and *wave5*, as well as *stereo*, a multibaseline stereo benchmark from the CMU benchmark suite [9] that operates on three 256 by 240 integer arrays of image data. The number of enabled cache ways is determined based on overall application cache characteristics, and therefore the number of enabled cache ways is only changed during context switches. Only L1 Dcache and L2 cache energy dissipations are calculated as the L1 Icache and main memory energy dissipations do not change significantly for these applications with the number of enabled L1 Dcache ways or with the L2 cache size.

The energy savings of selective cache ways depends on the amount of performance that can be traded off for energy. The *Performance Degradation Threshold (PDT)* signifies the average performance degradation relative to a cache with all ways enabled that is allowable for a given period of execution. If the PDT is 2%, and, for a given period of execution, performance is projected to degrade by 1% with three ways enabled, and 4% with two ways enabled, then three ways are enabled for that period of execution, so long as the total energy is less than that with all four ways enabled. This would not be the case if the extra misses with three ways enabled increase L2 cache energy more than the energy savings obtained with disabling one of the L1 Dcache ways. In this case, all four ways are enabled. In this study, the optimum number of enabled ways for each benchmark is determined from comparing



(a)



(b)

Figure 7: (a) Combined L1 Dcache and L2 cache energy savings and (b) actual performance degradation as a function of the performance degradation threshold.

performance and energy dissipation results. In an actual system, a profiling compiler or a runtime system such as Compaq's DCPI [4] can read cache hierarchy performance counter information and make assessments based on knowledge of relative L1 and L2 cache energy dissipations.

Figure 7 shows the energy savings and actual performance degradation incurred across all benchmarks as a function of the PDT. The energy savings is calculated from the average energy dissipation of all benchmarks with all ways enabled, and the average with the number of disabled ways allowable for a given PDT value. The performance degradation is similarly calculated from the corresponding IPC results. The actual performance degradation incurred is significantly less than the PDT value. Overall, roughly a 40% cache hierarchy energy savings is realized with less than a 2% performance degradation for a 512KB L2 cache. The benefits are less, yet still significant, for larger L2 caches due to the higher energy dissipated servicing an L1 Dcache miss. Even with a large 2MB on-chip L2 cache, a 25% energy savings is obtained with less than a 2% performance degradation using this CAP technique.

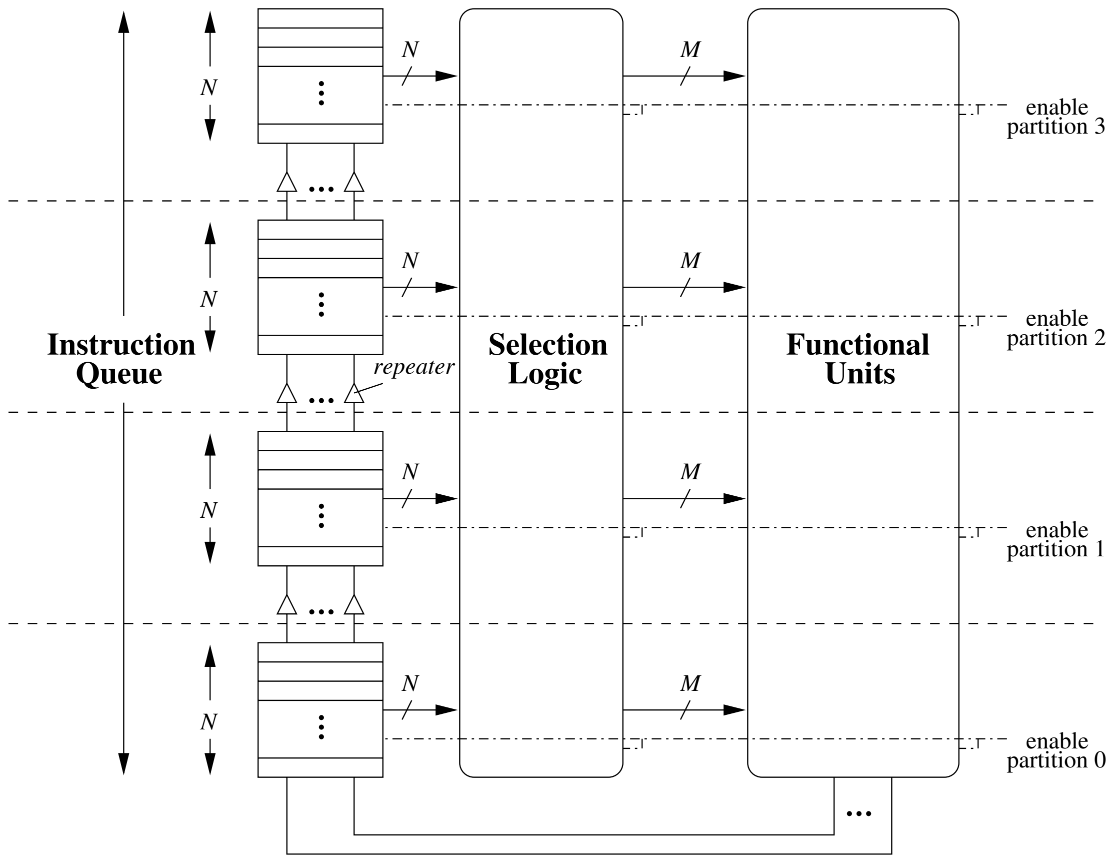


Figure 8: Partitioning the RUU and Functional Units using repeaters to electrically isolate RUU partitions.

Optimizing Core Processor Performance and Energy

Earlier in this article it was demonstrated how the effectiveness of three different aggressive core processor models varies during the execution of six of the SPEC95 benchmarks. CAP techniques can be used to dynamically change the configuration of the processor to match changing application requirements. The core resources for instruction issue, execution, and commit, consisting of the RUU and execution units, are partitioned into four sections as illustrated in Figure 8. Each partition of the instruction queue portion of the RUU is N entries deep. Each of the entries in the enabled partition can become eligible for issue on the next cycle if both operands and a functional unit of the correct type are available. The height and thus delay of the selection tree [16] portion of the RUU, which each cycle selects an instruction to issue to a given Functional Unit from the available pool of instructions, varies depending on the number of RUU partitions enabled. Branches of the tree corresponding to disabled RUU entries are themselves disabled. Similarly, trees that correspond to a disabled Functional Unit are completely disabled.

Repeaters are placed between the instruction queue partitions to electrically isolate the wires that run the length of each RUU section [1]. With this organization, if the upper three partitions are disabled, the instruction queue delay is no longer constrained by the long wire delays to these upper partitions, and therefore the remaining partition can operate at a faster speed. Delays for other combinations of partitions scale similarly. By globally partitioning other core structures, as is done with the *full config*, *half config*, and *quarter config* models discussed earlier, then several processor *slices* are defined that vary in complexity and critical path delays. Therefore, the speed of the CPU core can be changed according to the number of slices that are enabled. It is assumed in this analysis that changes in the core speed necessitate a change in the clock speed of the chip.

Optimizing for Performance and Energy

With this sliced processor organization, performance can be improved over a static design by dynamically trading off core processor resource complexity and operational speed. Doing so improves processor efficiency by enabling the amount of hardware complexity that is effectively used by the application, which in turn improves energy efficiency. Thus, as a byproduct of dynamically optimizing the complexity-speed tradeoff for performance purposes, energy dissipation can be considerably reduced as well [2]. The advantage of this approach, as opposed to the alternative of optimizing the energy-delay product, is that performance does not suffer at the expense of reducing energy consumption, an important criteria in a high performance machine. For such an environment, an appropriate metric is the Energy Throughput Ratio (ETR) which is given by [6]:

$ETR = Power/Throughput^2$.

A lower ETR is desirable as it indicates a more efficient processor. The $Throughput$ is calculated as the average rate of committed instructions per second, while $Power$ is given by:

$Power = V_{DD}^2 \cdot f_{CLK} \cdot C_{EFF}$.

V_{DD} is the supply voltage, f_{CLK} the clock frequency, and C_{EFF} the effective switched capacitance, which is a function of the physical capacitance and the switching activity. This equation for power ignores leakage and short-circuit power, both of which are typically small in relation to the switching power in current CMOS circuits.

To quantify the combined improvements in performance and energy with this approach, a CAP architecture which can switch (based on profile information) between the *full config*, *half config*, and *quarter config* configurations is compared with *full config*, the organization which achieves the best overall performance for the six SPEC95 benchmarks, even when accounting for cycle time. Based on the work of Palacharla [16], the relative cycle time of *full config* is assumed 20% higher than that of *half config*, which in turn is 20% higher than that of *quarter config*. The calculations for $Power$ only take into account the processor core. The C_{EFF} for *full config* is assumed to be twice that of *half config*, which in turn is twice that of *quarter config*.

For the *tomcatv*, *fpppp*, and *turb3d* benchmarks, the best-performing configuration for each individual benchmark is selected for its entire execution period. For *ijpeg*, *hydro2d*, and *li*, the configuration is changed dynamically during execution on average every 180 million, 5 million, and 1 million instructions, respectively. With the fast mechanism described earlier,

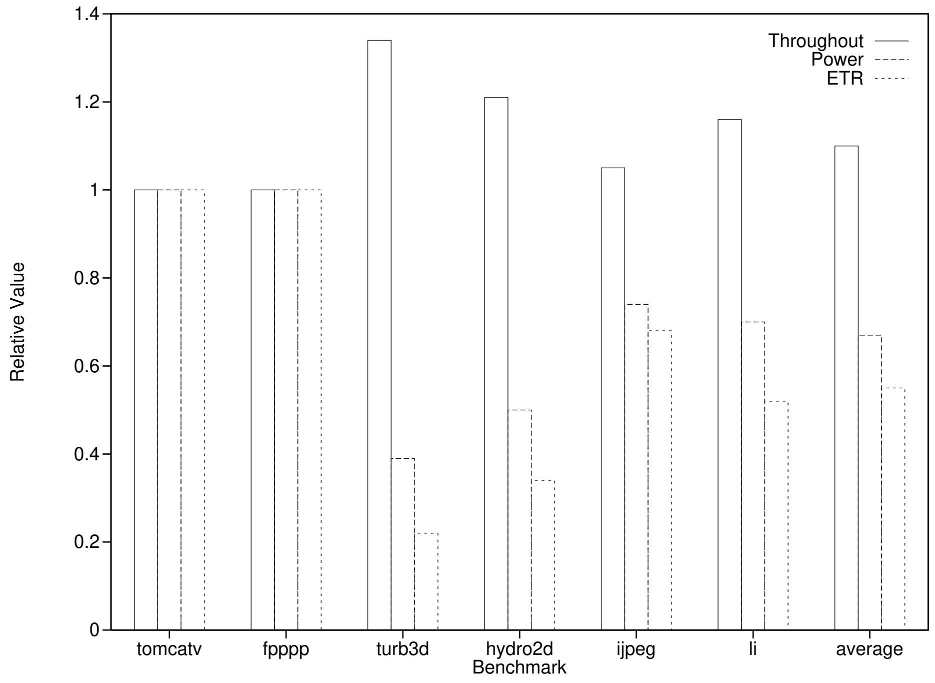


Figure 9: The Throughout, Power, and ETR of the CAP processor core relative to that of *full config*.

reconfiguration overhead is less than 0.1% for the fastest rate (*li*). However, when switching from a more aggressive to less aggressive configuration, instructions in an RUU partition that is to be disabled must be allowed to complete and commit their results. This can be handled by a two step reconfiguration process. First, the priority of instructions in these partitions is raised to allow these entries to empty quickly. In addition, no new instructions are placed in this partition. In the worst case situation of reconfiguring from *full config* to *quarter config*, this period would last about 30-40 clock cycles assuming the average IPC rate measured on the six SPEC95 benchmarks. However, during this period new instructions are still placed in the other RUU partition, and the processor is not idle as instruction execution otherwise continues normally. Once these partitions are emptied, they are disabled and the clock is changed accordingly.

Figure 9 shows relative values of throughput, processor core power, and processor core

ETR of the CAP configuration compared with *full config*. These values were measured from the steady-state behavior of all benchmarks, that is, after the initialization period. Both *tomcatv* and *fpppp* perform best with *full config* used throughout execution and so there is no difference for these benchmarks. On the other hand, *turb3d* performs best with *quarter config* always selected, and the result is a dramatic improvement in throughput, power, and ETR. The results for *hydro2d*, *ijpeg*, and *li* reflect the benefits of on-the-fly reconfiguration of hardware complexity during application execution. For example, for *hydro2d*, the ETR is 35% lower than would have been achieved in running its best overall configuration for its entire execution.

Overall, the CAP processor core produces a 33% reduction in power, and 45% improvement in processor efficiency (as calculated from relative ETR values) as compared to the conventional approach. A key point is that in contrast to many conventional low power techniques, this is achieved while *improving* processor throughout by 10%. This dual benefit is a product of the dynamic runtime optimization of hardware complexity and clock speed afforded by Complexity-Adaptive Processing.

Conclusions

Complexity-Adaptive Processing capitalizes on the natural partitioning of microprocessor hardware structures for speed purposes, and applies RTR techniques to dynamically improve both performance and energy efficiency. Two examples of CAP techniques demonstrate a significant improvement in both of these criteria. Our future plans include exploring CAP organizations with multiple clocked regions, and implementing a prototype chip using CAP memory hierarchy techniques in a leading industry architecture.

Acknowledgements

John Strasser coded the cache energy dissipation model, and to our knowledge, George Cai of Intel first coined the term “performance on demand.” This research is supported by the National Science Foundation under CAREER Award CCR-9701915 and grant CCR-9811929.

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] D.H. Albonesi. Selective cache ways: On-demand cache resource allocation. *Proceedings of the 32nd International Symposium on Microarchitecture*, November 1999.

[4] J. Anderson et al. Continuous profiling: Where have all the cycles gone? *Proceedings of the 16th Symposium on Operating Systems Principles*, October 1997.

[5] W.J. Bowhill et al. Circuit implementation of a 300-MHz 64-bit second-generation CMOS Alpha CPU. *Digital Technical Journal*, 7(1):100–118, Special Issue 1995.

[6] T.D. Burd and R.W. Brodersen. Energy efficient CMOS microprocessor design. *Proceedings of the 28th International HICSS Conference*, pages 288–297, January 1995.

[7] D. Burger and T.M. Austin. The SimpleScalar toolset, version 2.0. Technical Report TR-97-1342, University of Wisconsin-Madison, June 1997.

[8] J. Dean et al. ProfileMe: Hardware support for instruction-level profiling in out-of-order processors. *Proceedings of the 30th International Symposium on Microarchitecture*, pages 292–302, December 1997.

[9] P. Dinda et al. The CMU task parallel program suite. Technical Report CMU-CS-94-131, Carnegie Mellon University, March 1994.

[10] M.K. Gowan, L.L. Biro, and D.B. Jackson. Power considerations in the design of the Alpha 21264 microprocessor. *Proceedings of the 35th Design Automation Conference*, June 1998.

[11] J.L. Hennessy. Back to the future: Time to return to some long standing problems in computer systems? *Federated Computer Conference*, May 1999.

[12] M.B. Kamble and K. Ghose. Analytical energy dissipation models for low power caches. *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 143–148, August 1997.

[13] R.E. Kessler, E.J. McLellan, and D.A. Webb. The Alpha 21264 microprocessor architecture. *International Conference on Computer Design*, October 1998.

[14] A. Kumar. The HP PA-8000 RISC CPU. *IEEE Computer*, 17(2):27–32, March 1997.

[15] D. Matzke. Will physical scalability sabotage performance gains? *IEEE Computer*, 30(9):37–39, September 1997.

[16] S. Palacharla, N.P. Jouppi, and J.E. Smith. Quantifying the complexity of superscalar processors. Technical Report TR-96-1328, University of Wisconsin-Madison, November 1996.

[17] V. Tiwari et al. Reducing power in high-performance microprocessors. *Proceedings of the 35th Design Automation Conference*, June 1998.

[18] D.W. Wall. Limits of instruction-level parallelism. Technical Report 93/6, Digital Western Research Laboratory, November 1993.

[19] S.J.E. Wilton and N.P. Jouppi. An enhanced access and cycle time model for on-chip caches. Technical Report 93/5, Digital Western Research Laboratory, July 1994.

[20] 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.