Dynamic IPC/Clock Rate Optimization*

David H. Albonesi
Dept. of Electrical Engineering
University of Rochester
Rochester, NY 14627
albonesi@ee.rochester.edu

Abstract

Current microprocessor designs set the functionality and clock rate of the chip at design time based on the configuration that achieves the best overall performance over a range of target applications. The result may be poor performance when running applications whose requirements are not well-matched to the particular hardware organization chosen. We present a new approach called Complexity-Adaptive Processors (CAPs) in which the IPC/clock rate tradeoff can be altered at runtime to dynamically match the changing requirements of the instruction stream. By exploiting repeater methodologies used increasingly in deep sub-micron designs, CAPs achieve this flexibility with potentially no cycle time impact compared to a fixed architecture. Our preliminary results in applying this approach to on-chip caches and instruction queues indicate that CAPs have the potential to significantly outperform conventional approaches on workloads containing both general-purpose and scientific applications.

1 Introduction

Computer architects are constantly striving to design microarchitectures whose hardware complexity achieves optimal balance between instructions per cycle (IPC) and clock rate such that performance is maximized for a range of target applications. Although features such as wide issue windows and large L1 caches can produce high IPC for many applications, if a clock speed degradation accompanies the implementation of these large structures, the result may be lower performance for those applications whose IPC does not improve appreciably. These latter applications may perform better with a less-aggressive microarchitecture emphasizing high clock rate over high IPC. Thus, for a given set of target applications, there are seemingly endless combinations of features leading to different clock speeds and IPC values which achieve almost identical mean performance.

For example, the Digital 21164 [11] and HP PA-8000 [14] achieve almost identical SPECfp95 baseline results [1], yet each takes a much different approach to reaching this end. The 21164's streamlined in-order design and small (8KB) L1 caches (as well as aggressive implementation technology and circuit design) provides a clock rate that is roughly three times that of the PA-8000, while the PA-8000 provides a 56-entry out-of-order instruction window and multi-megabyte L1 caches (currently implemented off-chip). Although differences in instruction set architectures, process technologies, and compilers clearly factor into performance results, the *IPC/clock rate tradeoff* made in the design of key hardware structures can have a dramatic impact on individual application performance, with some applications favoring a simple, fast approach, and others performing better on a more complex design.

*This research was supported in part by NSF CAREER Award MIP-9701915.

Although the net result may be the same, with each sharing the lead in the SPECfp95 performance race, both of these implementations may suffer severe performance degradation on applications whose characteristics are not well-matched to the IPC/clock rate tradeoff point of the hardware design. This may be the case, for example, on the 21164 with applications with frequently-accessed megabyte-sized data structures that do not fit in the on-chip cache hierarchy, causing the processor to frequently run at the speed of the board-level cache. Such an application may perform better on the PA-8000 with its multi-megabyte L1 Dcache, even at its lower clock speed. Conversely, applications with small working sets and little exploitable instruction-level parallelism (ILP) may effectively waste the large Dcache and instruction window of the PA-8000, and run more efficiently on the faster 21164. Thus, diversity of hardware requirements from application to application forces microarchitects to implement hardware solutions that perform well *overall*, but which may compromise individual application performance. Worse yet, diversity may exist within an individual application. Wall found that the amount of ILP within an individual application varied during execution by up to a factor of three [27]. Thus, even implementations that are well-matched to the overall requirements of a given application may still exhibit suboptimal performance at various points of execution.

To address application diversity, proposed configurable architectures such as MATRIX [9] replace fixed hardware structures with reconfigurable ones in order to allow the hardware to dynamically adapt at runtime to the needs of the application. These *intrusive* approaches, however, may lead to decreased clock rate and increased latency, both of which may override the performance benefits of dynamic configuration. For these reasons, configurable architectures are currently relegated to specialized applications and have yet to be proven effective for general-purpose use.

This paper describes *Complexity-Adaptive Processors* (CAPs), a low-intrusive, *evolutionary* approach to implementing configurability within conventional microprocessors. CAPs are similar to complexity-*effective* processors [23] in that both try to obtain the highest performance by paying close attention to balancing IPC and clock speed in designing critical processor hardware. However, complexity-effective processors, like all conventional designs, fix the tradeoff between these two parameters at design time. CAPs employ configurable hardware for the core superscalar control and cache hierarchy structures of the processor in a way that minimizes, and often eliminates altogether, any cycle time or latency penalty relative to a conventional superscalar design, and a dynamic clock that adapts along with the hardware structure to allow each configuration to run at its full clock rate potential. Thus, CAPs provide many different IPC/clock rate tradeoff points within a single hardware implementation, yet do so while maintaining the high clock rate of a fixed architecture. If the configurable hardware can be effectively managed by selecting the

best configuration at critical runtime points, then a single CAP design may outperform conventional designs on a wide range of both scientific and general-purpose applications.

The rest of this paper is organized as follows. In the next section, we discuss scaling of logic and wire delays with feature size and the implications on the design of future processor hardware structures. We then describe in Section 3 how these trends will allow fixed structures to be made adaptive with little extra effort or delay penalty. Complexity-Adaptive Processors are introduced in Section 4, and in Section 5, we demonstrate how even simple implementations of complexity-adaptive cache hierarchies and instruction queues can significantly outperform conventional designs. In Section 6, we discuss the prospects for greater performance through finer-grained reconfigurable control of individual applications. Finally, in Section 7, we conclude and discuss future work.

2 Scaling Trends and Implications

Semiconductor feature sizes continue to decrease at a rapid pace, with 0.18 micron devices slated to appear in 1999, and 0.13 micron devices following in 2003 [2]. Although this creates greater transistor budgets for the microprocessor architect, because to a first order transistor delays scale linearly with feature size while wire delays remain constant, wire delays are increasingly dominating overall delay paths as feature sizes are decreased. For these reasons, repeater methodologies, in which buffers are placed at regular intervals within a long wire to reduce wire delays, are becoming more commonplace in deep submicron designs. For example, the Sun UltraSPARC-IIi microprocessor, implemented in a 0.25 micron CMOS process, contains over 1,200 buffers to improve wire delay [21]. Note that wire buffers are used not only in busses between major functional blocks, but within self-contained hardware structures as well. The forthcoming HP PA-8500 microprocessor, which is also implemented in 0.25 micron CMOS, uses wire buffers for the global address and data buses of its on-chip caches [12]. As feature sizes decrease to 0.18 micron and below, other smaller structures will require the use of wire buffers in order to meet timing requirements [18].

Our interest lies in the extent to which wire buffers will be required in the critical hardware structures of future dynamic superscalar processors. These include the instruction cache hierarchy, branch predictor, register rename logic, instruction queues and issue logic, register files, data cache hierarchy, Translation Lookaside Buffers (TLBs), and reorder buffers, as well the structures required for proposed new mechanisms such as value prediction [16]. These structures are primarily composed of RAM or CAM-based arrays implemented as replicated storage elements driven by global address and data buses. (Control signals may be required as well, which for the purpose of this discussion we include with the address bus.) If these structures are implemented with many elements or ports, their overall delay may reach the point where they impact processor cycle time.

In order to obtain a first-order understanding of general wire delay trends of these structures, we analyze both unbuffered and buffered wire delays of various sizes of caches and instruction queues using different feature sizes. We obtain these results using Bakoglu's optimal buffering methodology [4], technology and layout parameters from the CACTI cache cycle time model [28] (with parameters scaled appropriately for the feature size), and by assuming that buffer delays scale linearly with feature size while wire delays remain constant.

Figure 1 shows unbuffered and buffered cache address wire delays (data delays are identical) as a function of the number of subarrays and feature size for caches constructed from 2KB and 4KB cache subarrays. There is only one unbuffered curve as wire delays remain constant with feature size. These results indicate that 16KB and larger caches constructed from 2KB subarrays and implemented in 0.18 micron technology will benefit from buffering strategies. Using 4KB subarrays, a buffering strategy will clearly be beneficial for caches 32KB and larger with 0.18 micron technology.

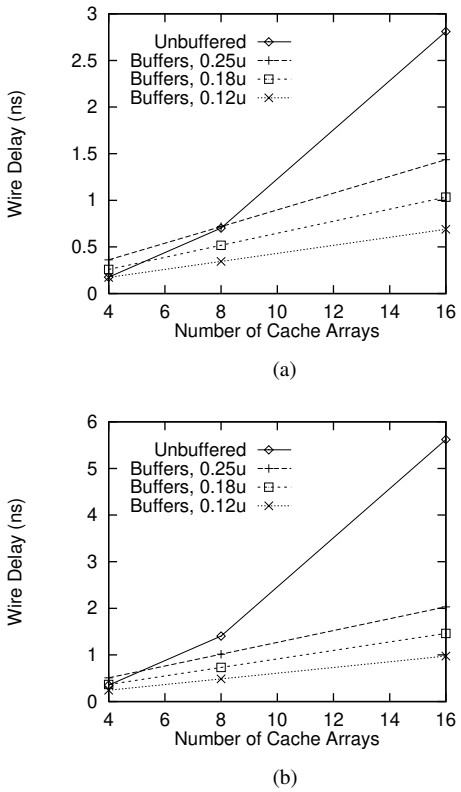


Figure 1: Cache wire delay as a function of the number of subarrays and technology using (a) 2KB and (b) 4KB cache subarrays.

Our instruction queue follows the R10000's integer queue design, each entry of which contains 52 bits of single-ported RAM, 12 bits of triple-ported CAM, and 6 bits of quadruple-ported CAM [29]. We assume that the area of a CAM cell is twice that of a RAM cell [20], and that the area grows quadratically with the number of ports, since both the number of wordlines and bitlines scale linearly with the number of ports. Under these assumptions, each R10000 integer queue entry is equivalent in area to roughly 60 bytes of single-ported RAM.

Figure 2 shows unbuffered and buffered integer queue wire delays as a function of the number of queue entries and feature size. Buffering performs better for a 32-entry queue with 0.12 micron technology, while larger queue sizes clearly favor the buffered approach with a feature size of 0.18 microns. As architects attempt to issue eight or more instructions in a cycle, integer queues larger than 32 entries will become necessary. In addition, increased issue width will necessitate an increase in the size of each queue entry, due to more ports and a widening of the fields for rename registers and levels of branch speculation. For these reasons, the use of

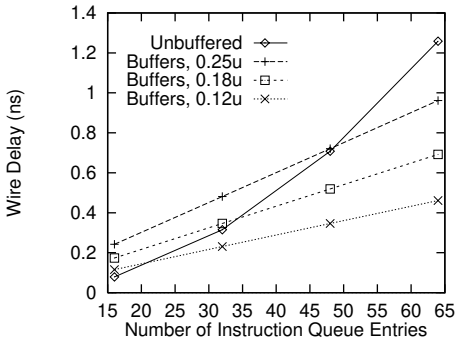


Figure 2: Integer queue wire delay as a function of the number of entries and technology.

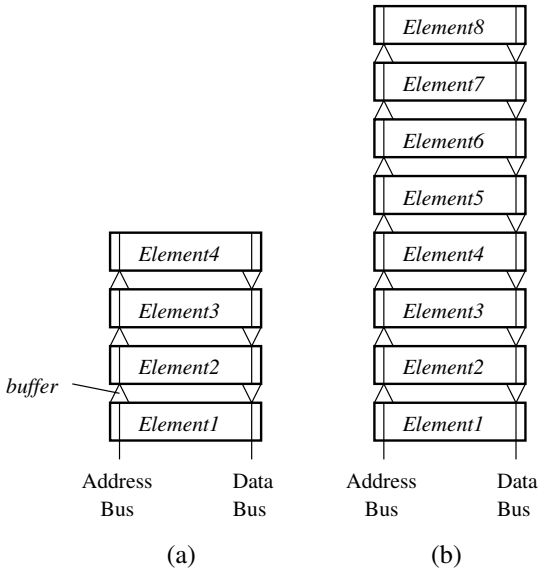


Figure 3: (a) A four-element hardware structure with wire buffers to reduce global address and data bus delays. (b) The same basic structure but with twice the number of elements.

buffers will clearly become necessary to minimize integer queue wire delays in future technologies.

Many other regular RAM or CAM-based structures, such as branch predictor tables and TLBs, may easily exceed these integer queue sizes, making them prime candidates for wire buffering strategies as well. Based on these initial results and the trends indicated by the UltraSPARC-IIi and PA-8000 designs, we believe that the use of buffered wire methodologies will within the next few years become standard practice in designing many of the critical superscalar control and cache structures of conventional microprocessors. The widespread adoption of these structures throughout the microprocessor creates new opportunities for architectural innovation as we explain in the next section.

3 A New Opportunity: Adaptive Hardware Structures

Figure 3(a) shows a conventional hardware structure containing four identical elements and three sets of buffers in both the global address and data busses to reduce wire delay. (Three is simply a convenient number for drawing purposes; the optimal number of buffers may actually be more or less than the number of elements.) Unlike unbuffered busses in which the entire line capacitance is seen at every element, each wire segment in Figure 3(a) has virtually equal delay and is electrically isolated from adjacent segments by the buffers. This isolation creates a distinct hierarchy of delays, in which individual element delays become independent of the number of remaining elements in the structure. The result is that the Element delays in Figure 3(a) are the same as that in Figure 3(b), despite the fact that the size of the structure has been doubled. This property has a profound impact on our ability to incorporate adaptivity in a low-intrusive manner, as we explain further below.

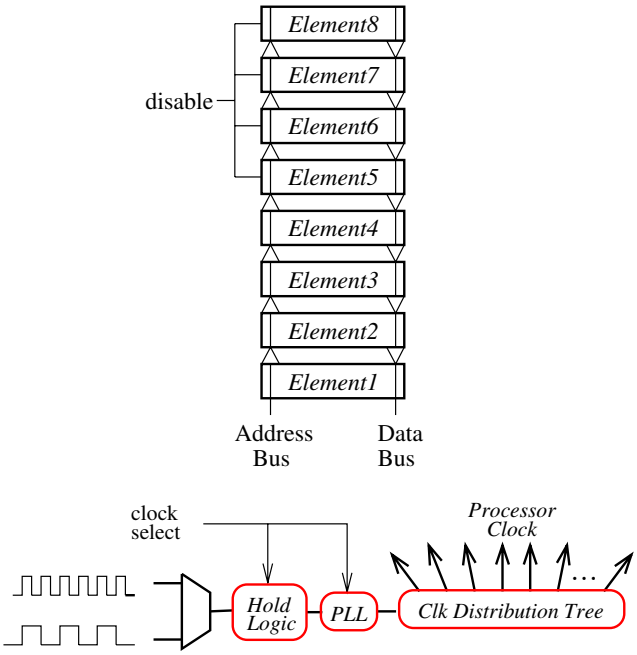


Figure 4: A hardware structure which can be configured with either four or eight elements.

If this particular structure is on the critical processor timing path when it contains four or more elements, then the implementation of the four-element structure of Figure 3(a) will result in a faster clock than the eight-element design of Figure 3(b). Because the latter structure is larger, its use may result in a higher IPC than the faster structure for applications which make frequent use of the additional elements. We can in fact make *both* of these options available in a single implementation if we start with the eight-element design, provide the ability to conditionally disable Elements 5-8, and employ a *dynamic clock* design which sets the on-chip processor clock speed according to whether four or eight elements are enabled. The resulting complexity-adaptive hardware structure, shown in Figure 4, can be dynamically reconfigured to meet the changing requirements of the instruction stream, yet do so with potentially no cycle time degradation relative to the fixed designs of Figure 3. We can generalize this concept to create structures with a finer *configuration increment*, which is the granularity at which the structure increases or decreases in size and/or other organizational parameter. The structures in Figure 3 can support a configuration increment of a single element if individual disable

signals are provided for each element. In general, the minimum configuration increment that can be supported without imposing a delay penalty is that which results from using the optimal number of buffers needed to minimize wire delay. In this case, if the conditional disabling logic (which may be necessary in a conventional design to reduce power dissipation) does not unduly impact timing delays, then the complexity-adaptive hardware structure will suffer no cycle time penalty relative to a conventional design, and in fact the two may be identical. Thus, once a repeater methodology has been adopted for a particular hardware structure for delay reasons, this fixed structure can be converted into a complexity-adaptive one with little or no additional effort. Pure logic structures that are designed by replicating common elements, such as a tree of priority encoders used in selecting instructions to issue [22], can also be made complexity-adaptive by selectively disabling particular elements. The flexibility of the CAP approach allows the designer to increase the size of certain structures to meet the requirements of particular "degenerate", but strategically important, applications, while allowing structures to be "shrunk" to better match the IPC/clock rate tradeoff point of more "average" applications. This flexibility, if managed properly, can afford a significant performance advantage over current fixed structures for many applications.

3.1 Varying Latency Instead of Clock Rate

To achieve high performance, it is essential that some structures be accessed in a single cycle. An example is an instruction queue, where a multi-cycle issue operation would severely increase data dependency-related stalls. Thus, it is appropriate when increasing the size of such structures to the point where they fall on the critical timing path to decrease the clock rate, if the increase in structure size results in a net performance gain.

For other structures where single-cycle access is not as critical, like the Dcache, an alternative to slowing the clock is to increase the latency (in cycles). The advantage of this alternative is that only instructions which make use of the hardware structure are affected, as the clock speed is not changed. Thus, in the case of the Dcache, arithmetic units are unaffected by the increase in Dcache size, and thus arithmetic instructions still complete at the same rate. However, careful design and location of selectable latches is necessary so as not to unduly impact critical path delays and hardware structure size. In addition, pipeline control and scheduling complexities introduced by this scheme need to be examined. Our future research plans include investigating these issues and determining the appropriate option (changing the clock, changing the latency, or changing both) for each hardware structure.

4 Complexity-Adaptive Processors

Figure 5 shows the overall organization of a Complexity-Adaptive Processor. The processor, caches, and external interface consist of conventional fixed hardware structures (denoted FS) intermixed with complexity-adaptive structures (denoted CAS) like those described in the previous section. Fixed structures are employed where implementing adaptivity is unwieldly, will strongly impact cycle time, or is ineffective due to a lack of diversity in target application requirements for the particular structure.

The Configuration Manager controls the organization of each CAS and the clock speed of the entire processor at appropriate execution points. The various clock speeds are predetermined based on worst-case timing analysis of each FS and combination of CAS configurations. In this figure, we have shown several clock sources

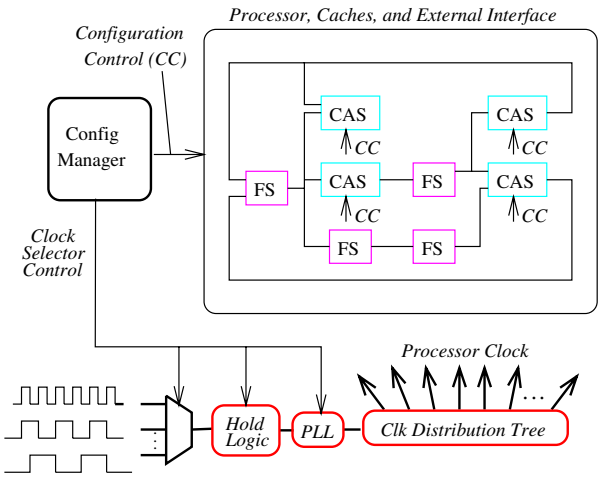


Figure 5: Overall organization of a Complexity-Adaptive Processor showing both fixed (FS) and complexity-adaptive structures (CAS). Configuration Control (CC) signals control the organization of each CAS.

selectable through a clock holding and multiplexing scheme (analogous to some scan designs), although other schemes are possible [8, 13].

Effective configuration management requires on-chip performance monitoring hardware, configuration registers, and good heuristics for selecting optimal CAS configurations and reconfiguration points during execution. These heuristics may be implemented in software, hardware, or both. A CAP compiler may perform profiling analysis to determine at which points within the application particular CAS configurations should be enabled. Alternatively, adaptive control hardware may read the performance monitoring hardware at regular intervals at runtime, analyze the performance information, predict the configuration which will perform best over the next interval (perhaps using modified branch-prediction techniques), and switch configurations as appropriate. Further analysis of these options is left as an area for future research. In this paper, we use a simple *process-level adaptive* scheme in which the configuration remains unchanged throughout the execution of a given application, but can vary from application to application.

4.1 Advantages of the CAPs Approach

The primary advantage of the CAPs approach is that it allows microarchitects to push hardware structures beyond cycle time limits to meet the needs of particular applications, without penalizing other applications that perform better with a smaller structure running at a faster clock rate. As issue widths increase, the complexity of larger superscalar control structures and on-chip caches is becoming more of a concern, due to the potential impact on cycle time [22]. Yet for some applications, increased cycle time can be justified if accompanied by a larger increase in IPC. A CAP design can be tailored at runtime to the optimal IPC/clock rate tradeoff point of the dynamic instruction stream, thereby achieving balanced performance across a wide range of applications.

A second advantage of the CAPs approach is that it is a technique that can be used with other advanced architectural approaches such as Superspeculative [17] and Trace processors [25]. The performance of these approaches is highly dependent on the effec-

tiveness of critical control and storage structures, which are typically largely RAM or CAM-based and therefore can be made complexity-adaptive. Thus, the adoption of a complexity-adaptive approach does not in general preclude other architectural innovations.

In addition to performance benefits, CAPs offer the potential for improved power management. The controllable clock frequency and hardware disables of a CAP design provide several performance/power dissipation design points that can be managed at runtime. The lowest-power mode can be enabled by setting all complexity-adaptive structures to their minimum size, and selecting the slowest clock. The processor can be put into this mode under certain conditions, for example when a power failure occurs and a uninterruptible power supply is enabled. In addition, a single CAP design can be configured for product environments ranging from high-end servers to low power laptops.

Another advantage is that complexity-adaptive structures can be easily implemented in asynchronous processor designs. These designs utilize a handshaking protocol instead of a global clock, with each stage potentially communicating at a different speed. With a complexity-adaptive approach, very large structures can be designed, yet the average stage delay can be much lower than the worst-case delay if faster elements are frequently accessed. Thus, stage delays are automatically adjusted according to the location of elements, obviating the need for a Configuration Manager.

An additional advantage is fast reconfiguration time relative to many other configurable architectures, due to implementation in custom silicon technology and the simple register and disable-based configuration scheme. However, the need to reliably switch clock sources may require tens of cycles to pause the active clock and enable the new clock. This is still much faster than many configurable architectures, considering the clock speed differential.

4.2 Potential Design Issues

At a first glance, perhaps the most radical aspect of a CAP is its dynamic clock. Commercial microprocessors are developed using established synchronous design techniques that assume a stable clock signal. However, these techniques can still be used in a CAP design, as a single clock distribution tree is used as with a conventional design. We currently view the front-end clock selection logic as an extension of some scan designs that introduce a second clock into the system, and reliably stop one clock and start another without loss of data. Note that other dynamic clocking schemes have been proposed [8, 13, 19], although these have not been implemented to our knowledge. Thus, we believe that a reliable, high performance flexible clocking system is viable, but more analysis is needed in this area.

Even if a dynamic clock is feasible, and no clock speed penalty is incurred, CAPs will likely be less efficient for most configurations than a fixed design with the same hardware features. This is because CAP designers are forced to define pipeline stages according to the fastest clock speed used. The resulting design may not be the most efficient for slower clock speeds, *e.g.*, branch misprediction penalties may be higher than that in a corresponding fixed design. Designers can circumvent this issue to some degree by increasing the minimum size of each CAS structure and using coarser grained increments. However, this restricts flexibility and so the right balance must be found. Based on our initial results, we believe that the performance advantages of CAPs will far override any lack of efficiency that may exist due to its flexible architecture.

Another inefficiency that may arise in a CAP design regards use of chip resources. Disabled portions of complexity-adaptive structures represent wasted transistors. However, conventional microprocessors do not always make the best use of resources as well. Furthermore, in some cases, we may be able to avoid disabling elements by instead using them as "backups" to the primary elements. For example, a complexity-adaptive cache can be divided into L1 and L2 sections as we describe in the next section. Branch predictor tables and TLBs may consist of single and two cycle lookup elements. An instruction queue may be divided into elements that are close to issuing in the following cycle ("on-deck" [24]), and a backup section of instructions waiting for all of their operands or for a long-latency operation to be completed. These can be transferred to the on-deck section once most of their operands become available or the resource nearly becomes free. Although a backup strategy may allow for more efficient silicon usage and higher IPC, the control functionality required may become quite complicated in some cases.

Several issues complicate the design of the Configuration Manager. First, in transitioning from one configuration to another, some "cleanup" operations may be required. For example, entries in the portion of a configurable queue that are to be disabled need to be emptied before reconfiguration takes place. In general, these operations are simple and have low enough overhead to not unduly impact performance. A second challenge regards the determination of the optimal reconfiguration frequency, a tradeoff between maintaining processor efficiency and minimizing reconfiguration overhead. A more complicated issue is that of next-configuration prediction when complexity-adaptive structures are implemented throughout the processor. Because of the amount of performance information that must be gleaned, and the interactions between different hardware structures, predicting the best-performing configuration for the next interval of operation can be quite complex.

A final issue involves communication between CAPs operating at different frequencies, which will likely necessitate implementing synchronizers between the processor clock and a fixed-frequency external bus clock. This approach is commonly used in multiprocessors that permit the coexistence of different microprocessors running at different clock speeds. Direct interfaces to external memory chips can be handled within the processor clock domain by making control signals programmable, as is common practice today to allow processors to take advantage of faster components when they become available.

5 Implementations of Complexity-Adaptive Techniques

In this section, we perform a preliminary evaluation of implementations of complexity-adaptive two-level on-chip Dcaches and instruction queues. We chose these two structures for our initial investigation as their design can have a large impact on overall performance. We emphasize that we have only briefly explored the design space of each of these areas, and thus the implementations, and our results, are very likely to be suboptimal. We expect that a more rigorous investigation will yield much greater performance improvements than we achieve in this initial investigation.

5.1 Methodology

We used Atom [26] to obtain address traces of the first 100 million data cache references of 21 applications: the entire SPEC95 benchmark suite (except for go which we had difficulty instrumenting), airshed, stereo, and radar from the CMU suite [10], and the NAS benchmark appcg [3]. We chose these applications to demonstrate how a complexity-adaptive design can outperform a conventional approach on SPEC95, while achieving a significant performance improvement on scientific applications which

may not perform as well on a general purpose machine. We ran each trace into a two-level cache simulator that assumed blocking caches and ignored access conflicts. We assumed a 4-way issue processor whose pipeline in the absence of L1 Dcache misses was 67% efficient (2.67 IPC).

Individual cache increment delays were found using CACTI [28] and scaling results to 0.18 micron technology using CACTI's scaling factor parameter. Global address and data bus delays were determined using Bakoglu's optimal buffering methodology [4]. The optimal number of buffers to minimize wire delays provided the necessary isolation to support our increment size. Whenever buffered line delays were faster than unbuffered delays, we used buffered values for the conventional cache hierarchy as well. We assumed that the L1 cache cycle time determined the cycle time of the processor, and used a three cycle L1 cache latency. L2 hit latencies were calculated as $\lceil (L2accesstime/L1cycletime) \rceil$, and the average L2 cache miss latency was 30ns, or 2-3 times the L2 hit latency, an estimate of the average latency with a large board-level cache.

We independently assessed the performance of complexity-adaptive instruction queues using the same applications (with the addition of go) and the SimpleScalar Toolset [7] for modeling 8-way out-of-order superscalar designs. We ran each benchmark for the first 100 million instructions, and assumed a perfect branch prediction mechanism, plentiful functional units, and perfect caches.

In analyzing instruction queue timing, we assumed that the instruction queue wakeup and selection logic is on the critical timing path for all configurations. (This assumes that bypass delays are reduced via clustering, which would produce a slightly lower IPC than our simulation model [23].) The wakeup logic determines which instructions within the instruction queue are ready to issue, while the select logic selects instructions to issue based on the available pool. The combined operation must be performed atomically in one cycle to allow dependent instructions to issue in successive cycles. We used Palacharla's 16-entry queue wakeup delay values [22] for 0.18 micron technology, and assumed that the operand tag lines are buffered between each group of 16 queue entries (our increment size). Our analysis indicates that for 0.18 micron technology, this produces roughly equal or lower delay than an unbuffered approach for all of our queue sizes. We then used the appropriate selection logic delay values from [22] based on the height of the selection tree required for each window size. (The selection logic in [22] is comprised of a tree of 4-bit priority encoders.) Thus, priority encoders associated with window entries that are not currently active are disabled, and the height and root node of the tree may vary as well.

For both cases, we use a simplified process-level adaptive configuration management approach in which we fix the complexity-adaptive organization for the duration of each individual application, and where the configuration registers are loaded/saved by the operating system on context switches. We assume that a CAP compiler and/or runtime environment can identify the best organization overall for an application. Because we disable unused instruction queue entries rather than use them as "backups" (as with the cache design), before we reconfigure to a smaller queue size, entries in the portion of the queue to be disabled must first issue. This low-overhead operation occurs only on context switches and therefore does not pose a noticeable performance penalty.

Because in each case all applications were run for the same number of instructions or Dcache references, we used the metric average time per instruction (TPI), computed as the cycle time divided by IPC, to evaluate overall performance and as the basis for reconfiguration decisions. For the cache design, we also measured the average TPI due to cache misses (TPI_{miss}) to evaluate how well our adaptive approach reduces miss penalties.

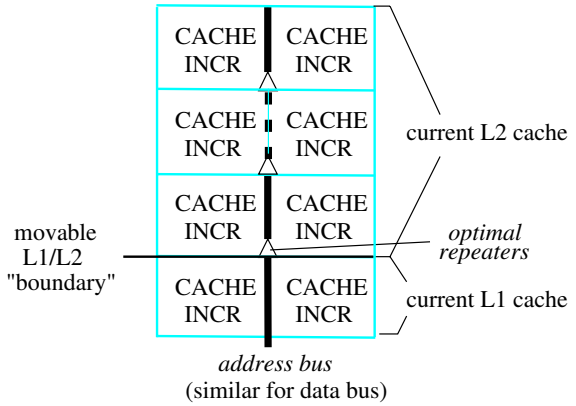


Figure 6: Structure of a complexity-adaptive cache hierarchy.

5.2 A Complexity-Adaptive Cache Hierarchy

Figure 6 shows the hardware structure of an on-chip complexity-adaptive Dcache hierarchy in which cache increments are assigned to the L1 and L2 Dcaches as needed by the application. The structure directly follows from large conventional cache designs that use multiple common cache banks [5, 6, 15]. Cache increments that are not part of the L1 cache are not disabled but are merely defined as belonging to the L2 Dcache. The "boundary" between the L1 and L2 caches is movable and defines the timing of the L1 and L2 caches. To simplify instruction scheduling and forwarding of load results, the L1 cache latency is kept constant in terms of cycles; the cycle time varies, however, depending on the delay of the slowest L1 cache increment. L2 cache banks have longer address and data bus delays than the L1 banks and therefore require more access cycles.

In order to reconfigure without having to invalidate or transfer data, we employ an exclusive caching policy within the hierarchy and enforce a simple rule: as an increment is added to (subtracted from) the L1 cache, its size *and associativity* are increased (decreased) by the increment size and associativity, and the L2 cache size and associativity are changed accordingly. This rule maintains a constant mapping of the index and tag bits independent of placement of the L1/L2 boundary. Two-level exclusive caching avoids the situation where two copies of the same cache block that were previously located separately in L1 and L2, are upon reconfiguration located in the same cache due to a redefinition of the L1/L2 boundary. With exclusion, a cache block is either in L1 or L2 but not both. The downside of our mapping rule is that it can lead to highly associative configurations. Although some applications would not benefit from such wide associativity, we believe that this is a reasonable tradeoff to simplify the mapping problem, and in practice we have found this solution to work well.

In contrast to a conventional cache in which the tag arrays are physically separated from the data arrays, each cache increment in Figure 6 is a complete *subcache* containing both tag and status bits. Grouping the tags and data arrays simplifies our design as it allows each cache to perform local hit/miss determination and enable its local data drivers on a hit. Because of exclusion and the way we have mapped the cache, only one of the L1 or L2 cache increments will have a hit on a given access. Thus, each cache can (in most cases) act locally without the need for global

information. An additional benefit of organizing the cache in this manner is shorter wire lengths between the tag comparators and data output drivers. Such localized wiring will become necessary with smaller feature sizes to maintain high clock rates [18].

5.2.1 Configurations Analyzed

In order to gain insight into the potential of our approach, we compared the performance of a single complexity-adaptive 128KB Dcache structure composed of 16 8KB two-way set associative and two-way banked increments against conventional two-level organizations of the same total size, with L1 sizes ranging from 8KB-64KB. We chose this design as it appeared to offer a better tradeoff between increment granularity and overall delay than a competing direct-mapped two-way banked 4KB increment design. Thus far, we have limited our investigation of this design to L1 caches up to 64KB in size.

5.2.2 Diversity of Cache Requirements

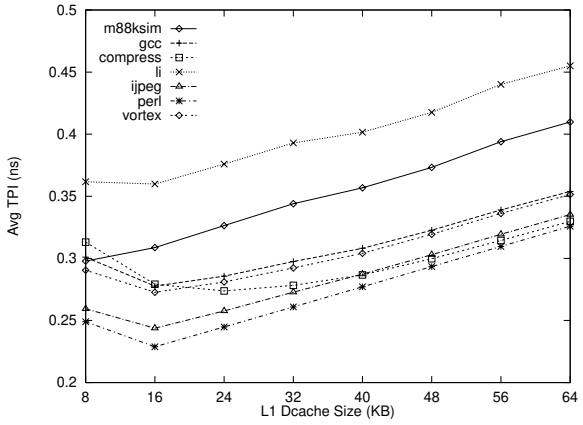
The performance improvement obtained with a process-level adaptive approach is highly dependent on the diversity of requirements from application to application. Figure 7 plots performance as a function of the L1 cache size with the L1/L2 boundary fixed throughout execution. The vast majority of the applications perform best with an 8KB or 16KB L1 Dcache, but this is not always due to small working set sizes. In some cases, some of the application's structures do not fit in our largest 64KB L1 Dcache, and so the configurations with the fastest cycle time perform best. For example, applu's L1 Dcache miss ratio is 9% with an 8KB L1 cache, and only drops to 8% with a 64KB L1 cache. Most of these misses miss in the L2 cache as well, indicating that our total cache size of 128KB is too small for this application.

Stereo and swim experience a large reduction in TPI as cache size is increased. Stereo's curve does not flatten out until the 48KB L1 cache point. Appcg experiences a sharp drop once L1 cache size is increased beyond 48KB. This is because of frequently-accessed data structures that require these larger caches to coexist. Of the integer applications, only compress improves with a cache larger than 16KB. Thus, although some applications benefit from the larger L1 caches, most perform best overall with small L1 caches, indicating that our workload is only modestly diverse in cache requirements from application to application.

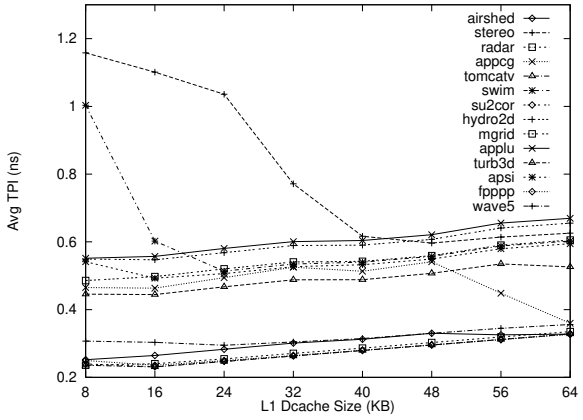
5.2.3 Overall Performance Results

Figures 8 and 9 plot average TPI_{miss} and average TPI, respectively, for the best-performing conventional configuration (which has 4-way set associative 16KB L1 Dcache) and the process-level adaptive case. Despite the fact that many of the applications perform best overall with the conventional organization, the simple process-level complexity-adaptive approach reduces TPI_{miss} by and average of 26% and delivers a respectable 9% average reduction in TPI over this configuration. Much of the benefit is due to significant improvements in stereo (reductions in TPI_{miss} and TPI of 65% and 46%), appcg (86% and 22%), swim (28% and 15%), and to a lesser extent, wave5, airshed, and radar.

The performance of stereo and appcg demonstrates how conventional microarchitectures that "design to the average" of a set of applications can exhibit significant performance degradation for applications that are not well-matched to the chosen hardware implementation. This may occur frequently when running applications with large data structures on general-purpose processors designed with good averaging-performing, but small, L1 Dcaches.



(a)



(b)

Figure 7: Variation of average TPI with L1 Dcache size for (a) integer and (b) floating point benchmarks. The L1/L2 boundary is fixed throughout execution.

In contrast, the adaptive approaches deliver good performance for each individual application, even bringing stereo's performance in line with the other applications. Note that our results indicate that significant gains can be obtained with the simple process-level approach of allowing each application to use one particular cache hierarchy configuration. Thus, even a simple CAP approach can afford a significant performance improvement over a conventional design.

The TPI_{miss} of the adaptive approach is in some cases higher than that of the conventional design. This is simply due to the fact that when optimizing for overall TPI, within some intervals, it is more beneficial to select a configuration with a faster clock than one with lower TPI_{miss}. For example, if there are very few load and store instructions, or if the L1 cache miss ratio is very low, then reducing miss activity will have little impact on overall TPI. For similar reasons, some applications with large overall reductions in TPI_{miss} gain very little benefit from a process-level adaptive approach. Compress, for example, achieves a 43% reduction in TPI_{miss} with the adaptive approach, but because loads and stores constitute less than 10% of the workload, this has only a minor impact on overall TPI.

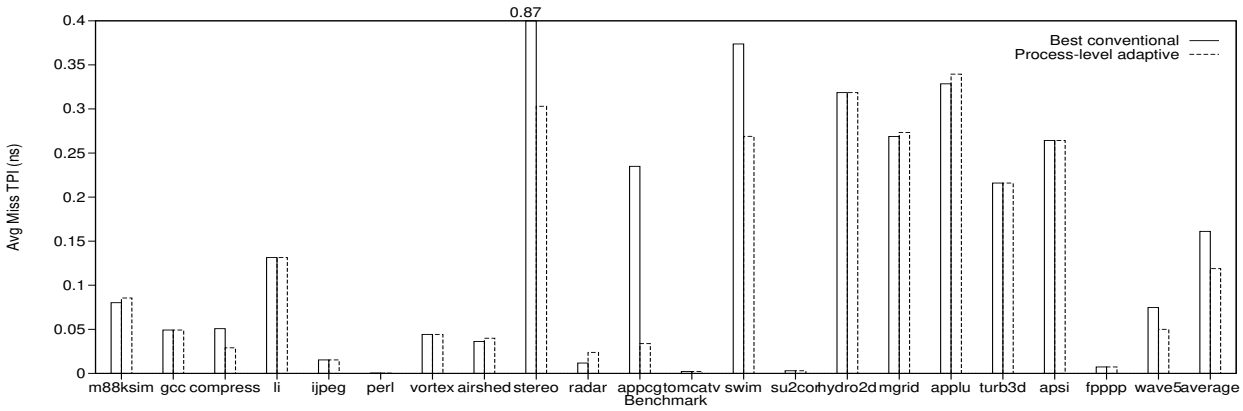


Figure 8: Average TPI_{miss} for conventional and process-level adaptive for each application and overall average.

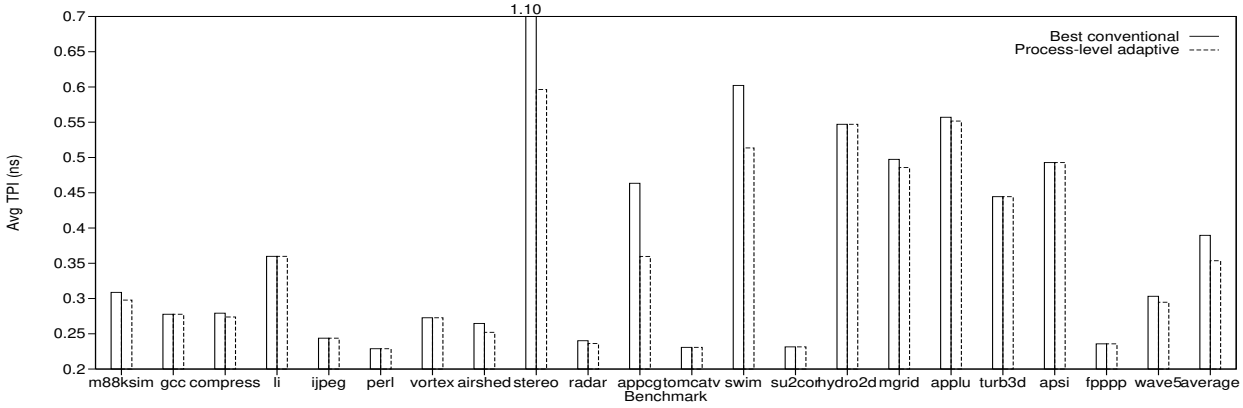


Figure 9: Average TPI for conventional and process-level adaptive for each application and overall average.

5.3 Complexity-Adaptive Instruction Issuing Logic

We compare fixed instruction queues ranging in size from 16 to 128 entries at 16-entry increments with a complexity-adaptive structure whose size could take on any of these values. We first examine application diversity, and then present overall performance results.

5.3.1 Diversity of Instruction Queue Requirements

Figure 10 shows diversity in instruction queue requirements from application to application. Most applications perform best with a the 64-entry instruction queue, although there are several exceptions. A 128-entry instruction queue performs best for compress, while radar, fpppp, and appcg clearly favor the smallest 16-entry configuration. For some applications, such as turb3d and vortex, there is a significant variation in requirements during execution, indicating that a finer-grain level of adaptivity may afford performance benefits over the simple process-level approach. We examine these opportunities more fully in Section 6.

5.4 Overall Performance Results

Figure 11 shows TPI results for the best conventional configuration (64-entry instruction queue) and the process-level adaptive approach. The process-level complexity-adaptive approach reduces TPI by an average of 7% over the conventional design. Much of the improvement is due to appcg (TPI reduction of 28%) and fpppp (21%). Appcg strongly favors a 16-entry configuration which puts its requirements far from the average. Radar, compress, and ijpeg achieve solid TPI reductions of 10%, 8%, and 8%, respectively, as well. Once again, although many of our applications favor the 64-entry design and therefore do not gain in performance, the few applications whose hardware requirements deviate from those in the conventional design perform significantly better under the simple process-level adaptive model.

5.5 Summary of Performance Results

Overall, we find these initial results to be quite encouraging for several reasons. First, in both cases, we explored a very small fraction of the design space, and thus we expect that a more rigorous investigation will yield much greater performance improvements. Second, our overall workload was only modestly diverse in its cache and instruction queue requirements, with many applications favoring the best conventional configuration in both cases. Third, the complexity-adaptive approach outperformed SPEC95 while dramatically improving performance for other applications that were not well-matched to the conventional organizations; for the cache design, the TPI of stereo was reduced by 46%, while appcg experienced a speedup of 28% with the complexity-adaptive instruction queue. Fourth, the fact that we achieved good performance improvements in applying the complexity-adaptive approach to two separate hardware structures indicates that these

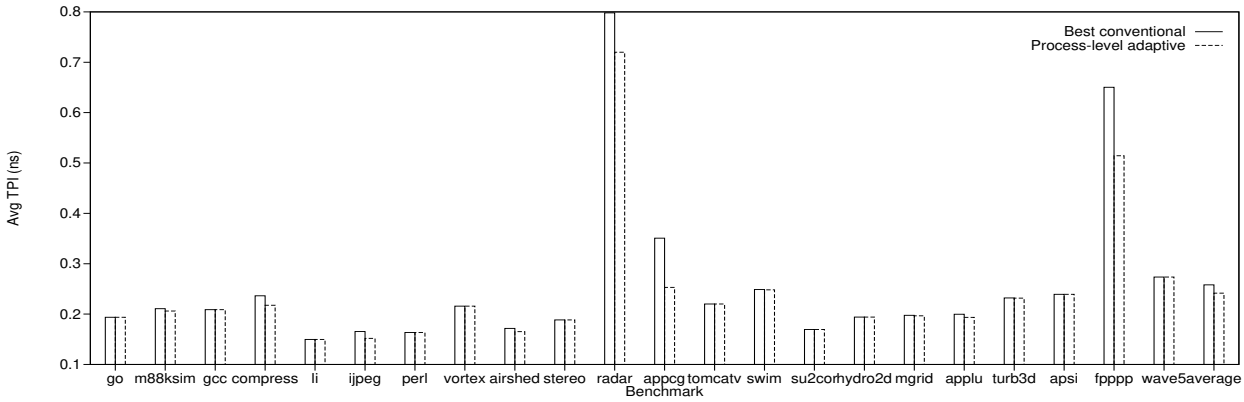
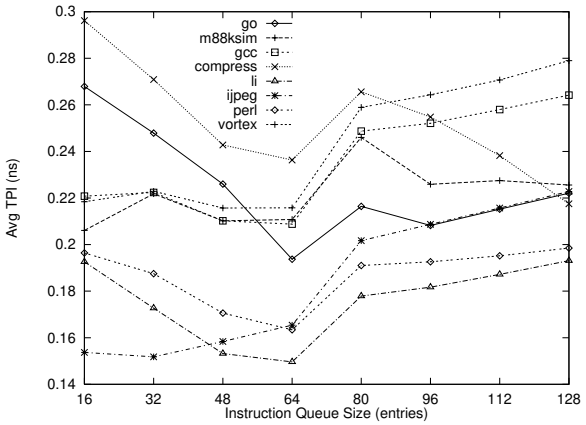
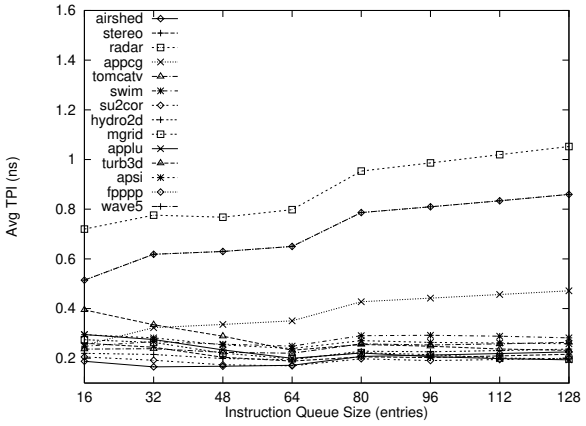


Figure 11: Average TPI for conventional and process-level adaptive for each application and overall average.



(a)



(b)

Figure 10: Variation of average TPI with the number of Instruction Queue entries for (a) integer and (b) floating point benchmarks.

techniques may be applied in concert to other critical parts of the machine (such as TLBs and branch predictors) to yield even greater performance improvements (although the number of configurations for a given structure might be limited due to larger delays in other structures). Finally, we used a simple process-level configuration management approach that does not exploit the diversity within individual applications. In the next section, we examine the potential for greater speedups by exploiting intra-application diversity.

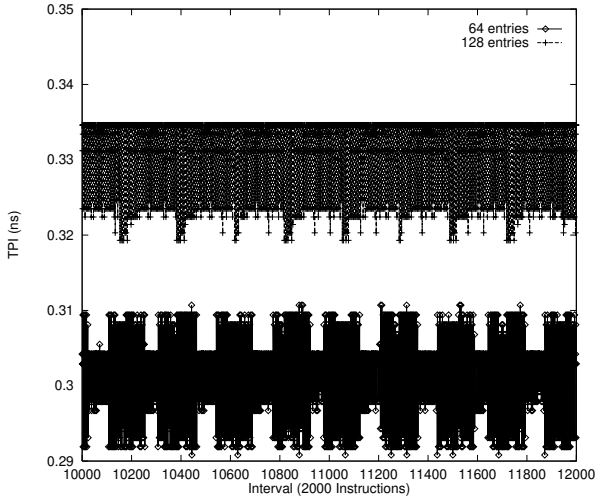
6 Prospects for Finer-Grained Adaptivity

Although several applications showed significant performance improvement with process-level adaptivity, many did not benefit as their average requirements matched those of the conventional configuration. In this section, we examine the intra-application diversity of two of these applications (vortex and turb3d) to determine if they can potentially benefit from reconfiguring during their execution. We chose these two applications as they are representative of the diversity we have observed thus far in several other applications (although some exhibit little intra-application diversity). For each, we compare performance snapshots of two instruction queue configurations that collectively perform best over 90% of the time: the 64 and 128-entry queue configurations for turb3d, and for vortex, the 16 and 64-entry configurations. Specifically, we are interested in long periods of execution during which one configuration performs best, and regular patterns that can be detected and exploited by a dynamic hardware prediction mechanism.

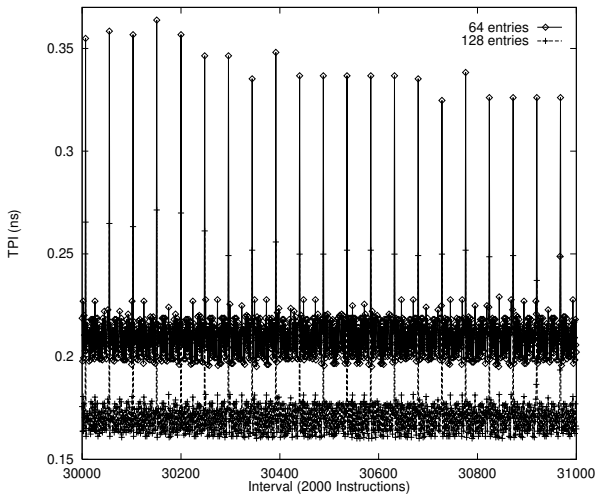
Figure 12 shows the average TPI for turb3d for the 64 and 128-entry queue configurations over two snapshots of execution. Each point in the plot represents the average TPI over an interval of 2000 instructions. Thus, Figure 12(a) shows four million instructions in all. Such long periods of execution in which one configuration clearly performs best are found throughout much of turb3d's execution, making its intra-application diversity easy to exploit.

Vortex exhibits periodic behaviors like those shown in Figure 13(a), in which the best-performing configuration alternates roughly every 15 intervals in a fairly regular fashion, indicating that the same instruction sequences are being encountered repeatedly. Such regular patterns can potentially be detected and exploited by a dynamic hardware predictor.

Not all of vortex's behavior is easy to predict, however. Figure 13(b) shows a sequence in which the best-performing configuration varies frequently and almost randomly. Note, however,

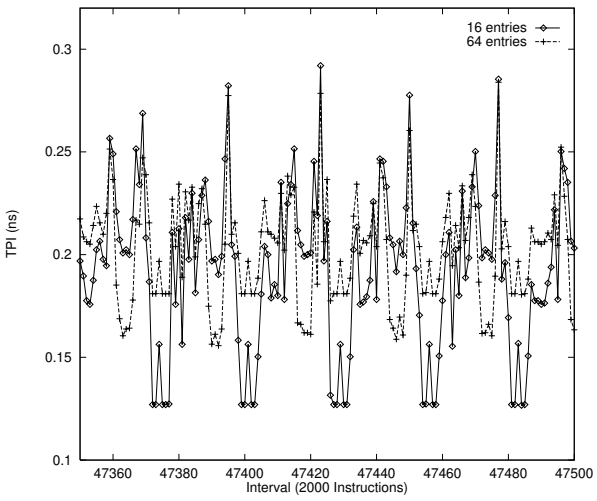


(a)

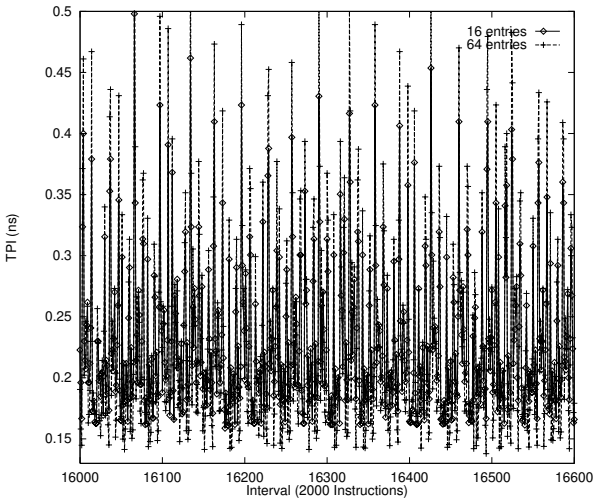


(b)

Figure 12: Two snapshots of turb3d's execution. In (a), the 64-entry configuration performs about 10% better overall, while in (b) the 128-entry performs about 20% better.



(a)



(b)

Figure 13: Two snapshots of vortex's execution. In (a), the best-performing configuration alternates in a regular pattern, while little predictability is observed in (b).

that the average performance of both configurations is about the same over this period. In this case, it would be better to maintain one or the other configuration over this entire period of execution rather than pay the penalty to reconfigure for no gain. Thus, as with value prediction, a complexity-adaptive hardware predictor should assign a confidence level to each prediction that is made, in order to avoid needless reconfiguration overhead.

7 Conclusions and Future Work

In this paper, we have presented Complexity-Adaptive Processors (CAPs), a new approach to incorporating configurability into commodity microprocessors in an low-intrusive manner. CAPs exploit the wire buffering methodologies being increasingly used as feature sizes decrease, to convert critical processor and cache hierarchy hardware structures into configurable ones with little or no cycle time impact. By employing a dynamic clock that allows each configuration to operate at its full clock rate potential, CAPs incorporate a number of different IPC/clock rate design points within a single implementation. This allows the hardware organization to be tailored to the needs of a diverse application base, while still maintaining the high clock rate of conventional fixed designs.

We discussed the design of two complexity-adaptive structures: a two-level on-chip cache hierarchy and an instruction queue. Using a simple configuration management strategy in which we maintained the same configuration throughout a given application's execution, we obtained average TPI reductions of 7-9% on an application base that was only modestly diverse. Individual applications whose requirements were not well-matched to the best overall-performing conventional design achieved reductions of up to 46%.

We finally explored the diversity within individual applications and discovered that the best-performing configuration often exhibited regular patterns of behavior that could potentially be exploited by a hardware-based prediction mechanism. We also discovered very irregular patterns suggesting that a confidence level should be assigned to predictions to avoid unnecessary reconfiguration overhead.

As we have explored a tiny fraction of the design space of this idea, much work remains. We need to expand our IPC and cycle time analysis environment and more thoroughly examine CAP design options for caches and instruction queues, as well as other structures such as TLBs and branch predictors, both individually and collectively. A more challenging task is to examine if complexity-adaptive techniques can be meaningfully applied to varying issue width and register file size. Dynamic clocking schemes need to be evaluated, and mechanisms to exploit intra-application diversity need to be explored, including an analysis of both hardware and software configuration management schemes. We remain optimistic that applying complexity-adaptive techniques widely throughout the processor, and exploring the design space of this approach, will lead to individual processor designs that can achieve excellent performance on a wide range of diverse applications.

Acknowledgements

The author wishes to thank the members of EE492 (Yehea Ismail, Xun Liu, Radu Secareanu, Patrick Furchill, and Justin Vlietstra) who helped flush out some of the initial ideas and developed the cache simulation tool, and Michael Scott and Tom Leblanc of the Computer Science department for providing the computing resources for this study.

References

[1] *Microprocessor Report*, 11(9):23, July 14, 1997.

[2] The national technology roadmap for semiconductors. *Semiconductor Industry Association*, 1997.

[3] D. Bailey et al. The NAS parallel benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, March 1994.

[4] H. Bakoglu and J. Meindl. Optimal interconnect circuits for VLSI. *IEEE Transactions on Computers*, 32(5):903–909, May 1985.

[5] P. Bannon. Private communication. June 1997.

[6] W. 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.

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

[8] M. Dean. STRIP: A self-timed RISC processor. Technical Report CSL-TR-92-543, Stanford University, July 1992.

[9] A. DeHon, D. Hartman, and E. Mirsky. MATRIX: A reconfigurable computing device with configurable instruction distribution. *Hot Chips IX Symposium*, August 1997.

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

[11] J. Edmondson et al. Internal organization of the Alpha 21164, a 300MHz 64-bit quad-issue CMOS RISC microprocessor. *Digital Technical Journal*, 7(1):119–135, Special Issue 1995.

[12] J. Fleischman. Private communication. November 1997.

[13] C. Georgiou, T. Larsen, and E. Schenfeld. Variable chip-clocking mechanism. *U.S. Patent number 5,189,314*, February 1993.

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

[15] G. Lesartre and D. Hunt. PA-8500: The continuing evolution of the PA-8000 family. *Proceedings of Compcon'97*, 1997.

[16] M. Lipasti and J. Shen. Exceeding the data-flow limit via value prediction. *Proceedings of the 29th International Symposium on Microarchitecture*, pages 226–237, December 1996.

[17] M. Lipasti and J. Shen. Superspeculative microarchitectures for beyond AD 2000. *IEEE Computer*, 30(9):59–66, September 1997.

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

[19] A. Merchant, B. Melamed, E. Schenfeld, and B. Sengupta. Analysis of a control mechanism for a variable speed processor. *IEEE Transactions on Computers*, 45(7):793–801, July 1996.

[20] J. Mulder, N. Quach, and M. Flynn. An area model for on-chip memories and its application. *IEEE Journal of Solid-State Circuits*, 26(2):98–106, February 1991.

[21] K. Normoyle et al. UltraSPARC-IIi: Expanding the boundaries of a system on a chip. *IEEE Micro*, 18(2):14–24, March 1998.

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

[23] S. Palacharla, N. Jouppi, and J. Smith. Complexity-effective superscalar processors. *Proceedings of the 24th International Symposium on Computer Architecture*, pages 206–218, June 1997.

[24] Y. Patt et al. One billion transistors, one uniprocessor, one chip. *IEEE Computer*, 30(9):51–57, September 1997.

[25] E. Rotenberg et al. Trace processors. *Proceedings of the 30th International Symposium on Microarchitecture*, pages 138–148, December 1997.

[26] A. Srivastava and A. Eustace. Atom: A system for building customized program analysis tools. *Proceedings of the SIGPLAN'94 Conference on Programming Language Design and Implementation*, June 1994.

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

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

[29] K. Yeager. The Mips R10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–41, April 1996.