# A Dynamically Tunable Memory Hierarchy

Rajeev Balasubramonian, David H. Albonesi, Alper Buyuktosunoglu, and Sandhya Dwarkadas

*Abstract*—
**The widespread use of repeaters in long wires creates the possibility of dynamically sizing regular on-chip structures. We present a tunable cache and translation lookaside buffer (TLB) hierarchy that leverages repeater insertion to dynamically trade off size for speed and power consumption on a per-application phase basis using a novel configuration management algorithm. In comparison to a conventional design that is fixed at a single design point targeted to the average application, the *dynamically tunable* cache and TLB hierarchy can be tailored to the needs of each application phase. The configuration algorithm dynamically detects phase changes and selects a configuration based on the application's ability to tolerate different hit and miss latencies in order to improve the memory energy-delay product. We evaluate the performance and energy consumption of our approach, and project the effects of technology scaling trends on our design.**

*Keywords*— **High performance microprocessors, Memory hierarchy, Reconfigurable architectures, Energy and performance of on-chip caches.**

## I. Introduction

MODERN microarchitectures continue to push the performance envelope by using architectural techniques to exploit improvements in technology. In the last 15 years, performance has improved at a rate of roughly 1.6 times per year with about half of this gain attributed to techniques for exploiting instruction-level parallelism and memory locality [19]. Correspondingly, however, the gap between processor speed and memory bandwidth and latency is continuing to increase. In addition, power dissipation levels have increased to the point where future designs may be fundamentally limited by this constraint in terms of the functionality that can be included in future microprocessors. The sheer number of transistors dedicated to the on-chip memory hierarchy in future processors (for example, roughly 92% of the transistors on the Alpha 21364 are dedicated to caches [8]) requires that these structures be effectively used so as not to needlessly waste chip power. Thus, new approaches are needed in order to prevent the memory system from fundamentally limiting future performance gains or exceeding power constraints. In this paper, we present a dynamically configurable cache and translation lookaside buffer (TLB) organization that exploits trends in technology to provide low-cost configurability in order to trade size for speed in the memory hierarchy.

The most common conventional memory system today is the multi-level memory hierarchy. The rationale behind this approach, which is used primarily in caches but also in some TLBs (*e.g.*, in the MIPS R10000 [36]), is that a combination of a small, low-latency L1 memory backed by a higher capacity, yet slower, L2 memory and finally by main memory provides the best trade-off between optimizing hit time and miss time. The fundamental issue with these designs is that they are targeted to the average application — no single memory hierarchy organization proves to be the best for all applications. When running a diverse range of applications, there will inevitably be significant periods of execution during which performance degrades and energy is needlessly expended due to a mismatch between the memory system requirements of the application and the memory hierarchy implementation. As an example, programs whose working sets exceed the L1 capacity may expend considerable time and energy transferring data between the various levels of the hierarchy. If the miss tolerance of the application is lower than the effective L1 miss penalty, then performance may degrade significantly due to instructions waiting for operands to arrive. For such applications, a large, single-level cache (as used in the HP PA-8X00 series of microprocessors [18], [23], [24]) may perform better and be more energy-efficient than a two-level hierarchy for the same total amount of memory.

In this paper, we present a configurable cache and TLB orchestrated by a configuration algorithm that tailors the memory hierarchy to each individual application phase in order to improve the performance and energy-efficiency of the memory hierarchy. Key to our approach is the exploitation of the properties of conventional caches and future technology trends in order to provide cache and TLB configurability in a low-intrusive and low-latency manner. Our cache and TLB are logically designed and laid out as a *virtual two-level, physical one-level* non-inclusive hierarchy, where the partition between the two levels is dynamic. The non-inclusive nature of the hierarchy minimizes the overheads at the time of repartitioning. Our approach monitors cache and TLB usage by detecting phase changes, and improves performance by resizing (using an exploration phase) the cache and TLB to properly balance hit latency intolerance with miss latency intolerance dynamically during application execution (using CPI as the ultimate performance metric). We evaluate the use of miss rates and branch frequencies to detect phase changes and limit the exploration space, as well as the use of more complex metrics that attempt to measure application hit and miss intolerance more accurately. Furthermore, instead of changing the clock rate as proposed in [2], we implement a cache and TLB with a variable latency so that changes in the organization of these structures only impact memory instruction latency and throughput. We also evaluate energy-aware modifications to the configuration algorithm that trade off a modest amount of performance for significant energy savings.

Our previous approaches to this problem ([2], [3]) have exploited the partitioning of hardware resources to enable/disable parts of the cache under software control, but in a limited manner. The issues of how to practically implement such a design were not addressed in de-

tail, the analysis only looked at changing configurations on an application-by-application basis (and not dynamically during the execution of a single application), and the simplifying assumption was made that the best configuration was known for each application. Furthermore, the organization and performance of the TLB was not addressed, and the reduction of the processor clock frequency with increases in cache size limited the performance improvement that could be realized.

We also expand on our more recent work ([6], [7]) in many ways. First, we examine how the results for our configurable memory hierarchy scale for additional processor frequencies beyond the base 1GHz frequency used exclusively in our earlier papers, and discuss how higher frequencies may simplify the timing of our design. Second, we evaluate the benefits of using more complex latency tolerance metrics for our phase detection and for limiting our search heuristics, and compare it with the simpler approach of using miss rates and branch frequencies. Finally, we quantify the energy-delay product benefits for our two energy-aware configurations in relation to the baseline conventional and configurable approaches.

The rest of this paper is organized as follows. The cache and TLB architectures are described in Section II including the modifications necessary to enable dynamic reconfiguration. In Section III, we discuss the dynamic selection mechanisms, including the counter hardware required and the configuration management algorithms. In Sections IV and V, we describe our simulation methodology and present a performance and energy dissipation comparison with conventional multi-level cache and TLB hierarchies for two technology design points and several clock frequencies. Finally, we discuss related work in Section VI, and we conclude in Section VII.

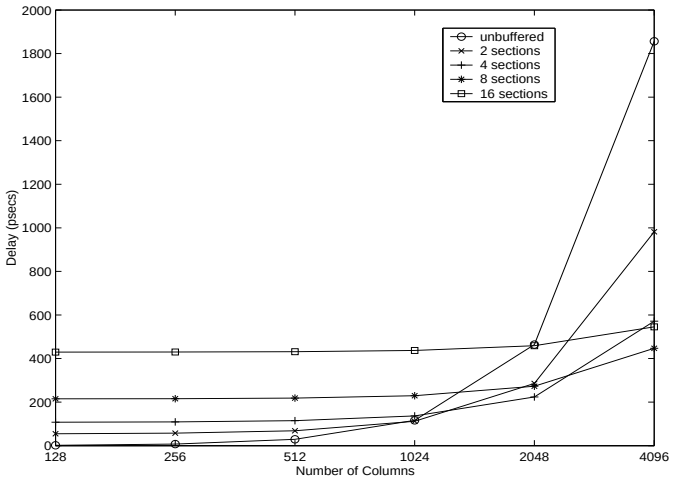



Fig. 1. Wordline wire delay as a function of the number of array columns for various numbers of wire sections.

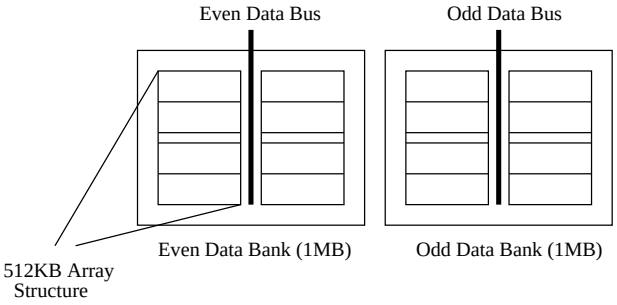



Fig. 2. The overall organization of the cache data arrays.

## II. Cache and TLB Circuit Structures

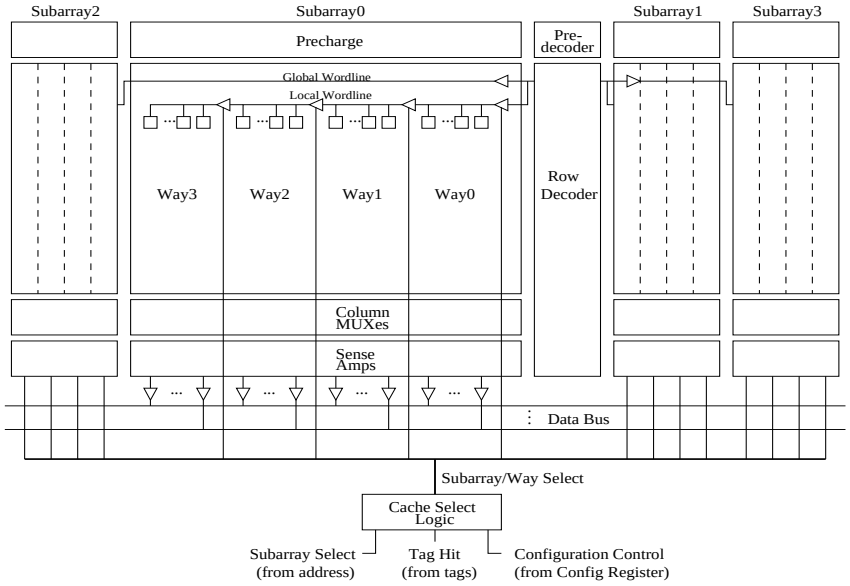
We present the design of a cache and TLB that can be dynamically partitioned in order to change size and speed. Our cache and TLB are logically designed as *virtual two-level, physical one-level* non-inclusive hierarchies, where the partition between the two levels is dynamic. The non-inclusive nature of the hierarchy minimizes the overheads at the time of repartitioning. In the following, we describe the circuit structures of the conventional caches and TLBs, and the modifications made to the conventional layout in order to accommodate the repartitioning with minimal impact on performance and area.

### A. Configurable Cache Organization

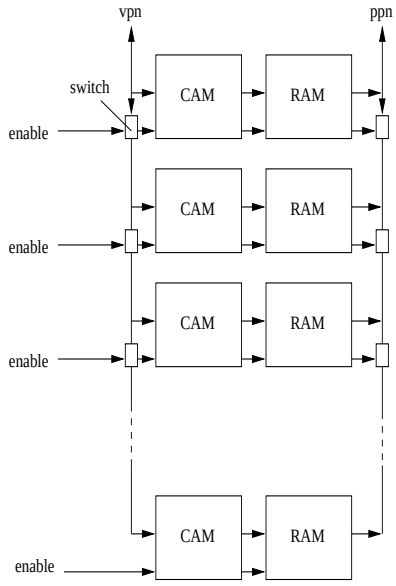
The cache and TLB layouts (both conventional and configurable) that we model follow that described by McFarland in his thesis [25]. McFarland developed a detailed timing model for both the cache and TLB that balances both performance and energy considerations in subarray partitioning, and which includes the effects of technology scaling.

We took into account several considerations in choosing the cache layout as well as parameters such as size and associativity for our configurable cache and the L2 cache in a conventional processor. First, we determined that the cache should be at least 1MB in size. We based this on the size of on-chip L2 caches slated to be implemented in next generation processors (such as the Alpha 21364 [8] which will have 1.5MB of on-chip cache). Based on performance simulation results with our benchmark suite, we picked 2MB as the target size for our configurable cache as well as for the L2 (or combined L1 and L2) of the conventional baseline memory hierarchies.

To further define the number of subarrays and associativity, we calculated (following Bakoglu [5]) the SRAM array wordline delay as a function of the number of array columns and the number of wire sections (separated by repeater switches) using the 0.1 $\mu$m parameters of McFarland [26]. The results are shown in Figure 1 for an unbuffered wire (no repeater switches), a wire broken into two sections by a single repeater switch, and so on. Note that repeater switches provide no delay benefit for arrays of 1024 columns or less. For the longer wordlines in arrays of 2048 columns or more, the "linearization" of the delay provided by repeater switches (as opposed to the quadratic delay increase of the unbuffered wire) results in a decrease in delay. The best delay is achieved with four repeater switches for




(a) Data array of the cache.




(b) TLB.

Fig. 3. (a) The organization of the data array section of one of the 512KB cache structures. (b) The organization of the configurable TLB.

2048 columns, and eight for 4096 columns.

Based on the above constraints, on delay calculations using various numbers of subarrays and layouts, and on the need to make the cache banked to obtain sufficient bandwidth, we arrived at the organization shown in Figure 2. The cache is structured as two 1MB interleaved banks[1]. In order to reduce access time and energy consumption, each 1MB bank is further divided into two 512KB SRAM structures (with data being block interleaved among the structures), one of which is selected on each bank access. We make a number of modifications to this basic structure to provide configurability with little impact on access time, energy dissipation, and functional density.

The data array section of the configurable structure is shown in Figure 3 (a) in which only the details of one subarray are shown for simplicity (The other subarrays are identically organized.). There are four subarrays, each of which contains four ways. Each of these subarrays has 512 rows and 2048 columns. In both the conventional and configurable cache, two address bits (*Subarray Select*) are used to select only one of the four subarrays on each access in order to reduce energy dissipation. The other three subarrays have their local wordlines disabled and their precharge, sense amp, and output driver circuits are not activated. The TLB virtual to real page number translation and tag check proceed in parallel and only the output drivers for the way in which the hit occurred are turned on. Parallel TLB and tag access can be accomplished if the operating system can ensure that *index_bits-page_offset_bits* bits of the virtual and physical addresses are identical, as is the case for the four-way set associative 1MB dual-banked L1 data cache in the HP PA-8500 [17].

In order to provide configurability while retaining fast access times, we implement several modifications to McFarland's baseline design as shown in Figure 3 (a):

- McFarland drives the global wordlines to the center of each subarray and then the local wordlines across half of the subarray in each direction in order to minimize the worst-case delay. In the configurable cache, because we are more concerned with achieving comparable delay with a conventional design for our smallest cache configurations, we distribute the global wordlines to the nearest end of each subarray and drive the local wordlines across the entire subarray.
- McFarland organizes the data bits in each subarray by bit number. That is, data bit 0 from each way are grouped together, then data bit 1, *etc.* In the configurable cache, we organize the bits according to ways as shown in Figure 3 (a) in order to increase the number of configuration options.
- Repeater switches are used in the global wordlines to electrically isolate each subarray. That is, subarrays 0 and 1 do not suffer additional global wordline delay due to the presence of subarrays 2 and 3. Providing switches as opposed to simple repeaters also prevents wordline switching in disabled subarrays thereby saving dynamic power.
- Repeater switches are also used in the local wordlines to electrically isolate each way in a subarray. The result is that the presence of additional ways does not impact the delay of the fastest ways. Dynamic power dissipation is also reduced by disabling the wordline drivers of disabled ways.
- *Configuration Control* signals from the *Configuration Register* provide the ability to disable entire subarrays or ways within an enabled subarray. Local wordline and data output drivers and precharge and sense amp circuits are not activated for a disabled subarray or way.

Using McFarland's area model, the additional area from adding repeater switches to electrically isolate wordlines is 7%. In addition, based on our column length (2048),

[1]The banks are word-interleaved when used as an L1/L2 cache hierarchy and block interleaved when used as L2/L3.

Subarray/Way Allocation (L1 or L2)

| Cache Configuration | L1 Size | L1 Assoc | L1 Acc Time | Subarray 2 | | | | Subarray 0 | | | | Subarray 1 | | | | Subarray 3 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | W3 | W2 | W1 | W0 | W3 | W2 | W1 | W0 | W0 | W1 | W2 | W3 | W0 | W1 | W2 | W3 |
| 256-1 | 256KB | 1 way | 2.0 | L2 | L2 | L2 | L2 | L2 | L2 | L2 | ***L1*** | ***L1*** | L2 | L2 | L2 | L2 | L2 | L2 | L2 |
| 512-2 | 512KB | 2 way | 2.5 | L2 | L2 | L2 | L2 | L2 | L2 | ***L1*** | ***L1*** | ***L1*** | ***L1*** | L2 | L2 | L2 | L2 | L2 | L2 |
| 768-3 | 768KB | 3 way | 2.5 | L2 | L2 | L2 | L2 | L2 | ***L1*** | ***L1*** | ***L1*** | ***L1*** | ***L1*** | ***L1*** | L2 | L2 | L2 | L2 | L2 |
| 1024-4 | 1024KB | 4 way | 3.0 | L2 | L2 | L2 | L2 | ***L1*** | ***L1*** | ***L1*** | ***L1*** | ***L1*** | ***L1*** | ***L1*** | ***L1*** | L2 | L2 | L2 | L2 |
| 512-1 | 512KB | 1 way | 3.0 | L2 | L2 | L2 | ***L1*** | L2 | L2 | L2 | ***L1*** | ***L1*** | L2 | L2 | L2 | ***L1*** | L2 | L2 | L2 |
| 1024-2 | 1024KB | 2 way | 3.5 | L2 | L2 | ***L1*** | ***L1*** | L2 | L2 | ***L1*** | ***L1*** | ***L1*** | ***L1*** | L2 | L2 | ***L1*** | ***L1*** | L2 | L2 |
| 1536-3 | 1536KB | 3 way | 4.0 | L2 | ***L1*** | ***L1*** | ***L1*** | L2 | ***L1*** | ***L1*** | ***L1*** | ***L1*** | ***L1*** | ***L1*** | L2 | ***L1*** | ***L1*** | ***L1*** | L2 |
| 2048-4 | 2048KB | 4 way | 4.5 | ***L1*** | ***L1*** | ***L1*** | ***L1*** | ***L1*** | ***L1*** | ***L1*** | ***L1*** | ***L1*** | ***L1*** | ***L1*** | ***L1*** | ***L1*** | ***L1*** | ***L1*** | ***L1*** |

Fig. 4. Possible L1/L2 cache organizations that can be configured shown by the ways that are allocated to L1 and L2. Only one of the four 512KB SRAM structures is shown. Abbreviations for each organization are listed to the left of the size and associativity of the L1 section, while L1 access times in cycles are given on the right. Note that the TLB access may dominate the overall delay of some configurations. The numbers listed here simply indicate the relative order of the access times for all configurations and thus the size/access time trade-offs allowable.

the use of repeater switches to isolate each way achieves a faster propagation delay than with unbuffered lines (Figure 1). Moreover, because local wordline drivers are required in a conventional cache, these extra buffers do not impact the spacing of the wordlines, and thus bitline length is unaffected. In terms of energy, the addition of repeater switches increases the total memory hierarchy energy dissipation by 2-3% (using the model described in Section IV) in comparison with a cache with no repeaters for the simulated benchmarks.

### *B. Configurable Cache Operation*

With these modifications, the sizes, associativities, and latencies of the resulting virtual two-level, physical one-level non-inclusive cache hierarchy are *dynamic*. That is, a single large cache organization serves as a configurable two-level non-inclusive cache hierarchy, where the ways within each subarray that are initially enabled for an L1 access are varied to match application characteristics. The latency of access is changed on half-cycle increments according to the timing of each configuration. Half cycle increments are required to provide the granularity to distinguish the different configurations in terms of their organization and speed. Such an approach can be implemented by capturing cache data using both phases of the clock, similar to the double-pumped Alpha 21264 data cache [22], and enabling the appropriate latch according to the configuration. The advantages of this approach is that the timing of the cache can change with its configuration while the main processor clock remains unaffected, and that no clock synchronization is necessary between the pipeline and cache/TLB.

However, because a constant two-stage cache pipeline is maintained regardless of the cache configuration, cache bandwidth degrades for the larger, slower configurations. Furthermore, the implementation of a cache whose latency can vary on half-cycle increments requires two pipeline modifications. First, the dynamic scheduling hardware must be able to speculatively issue (assuming a data cache hit) load-dependent instructions at different times depending on the currently enabled cache configuration. Second, for some configurations, running the cache on half-cycle increments requires an extra half-cycle for accesses to be caught by the processor clock phase.

The possible configurations for a 2MB L1/L2 on-chip cache hierarchy at 0.1$\mu$m technology are shown in Figure 4. Although multiple subarrays may be enabled as L1 in an organization, as in a conventional cache, only one is selected each access according to the *Subarray Select* field of the address. When a miss in the L1 section is detected, all tag subarrays and ways are read. This permits hit detection to data in the remaining portion of the cache (designated as L2 in Figure 4). When such a hit occurs, the data in the L1 section (which has already been read out and placed into a buffer) is swapped with the data in the L2 section. In the case of a miss to both sections, the displaced block from the L1 section is placed into the L2 section. This prevents thrashing in the case of low-associative L1 organizations.

The direct-mapped 512KB and two-way set associative 1MB cache organizations are lower energy, and lower performance, alternatives to the 512KB two-way and 1MB four-way organizations, respectively. These options activate half the number of ways on each access for the same capacity as their counterparts. For execution periods in which there are few cache conflicts and hit latency tolerance is high, the low energy alternatives may result in comparable performance yet potentially save considerable energy. These configurations are used in an energy-aware mode of operation as described in Section III.

Note that because some of the configurations span only two subarrays, while others span four, the number of sets is not always the same. Hence, it is possible that a given address might map into a certain cache line at one time and

into another at another time (called a *mis-map*). However, the non-inclusive nature of our cache helps prevent aliasing and guarantees correctness. In cases where subarrays two and three are disabled, the high-order *Subarray Select* signal is replaced by an extra tag bit. This extra tag bit is used to detect mis-maps. In order to minimize the performance impact of mis-mapped data, an L2 look-up examines twice as many tags as the conventional L2 in order to examine all subarrays where mis-mapped data may reside. Mis-mapped data is handled the same way as a L1 miss and L2 hit, *i.e.*, it results in a swap. Our simulations indicate that such events are infrequent.

In conventional two-level hierarchies, the L2 implements additional hardware to ensure cache coherence with the L2s of other chips. This involves replication of its tags and a snooping protocol on the system bus. The use of a non-inclusive L1-L2 does not affect this in any way. The tags for the entire 2MB of on-chip cache are replicated (just as for the 2MB L2 in the conventional hierarchy) and the snooping protocol ensures that data in the L1 and L2 are coherent with other caches. There is potential for added interference with the processor to L1 datapath when compared to a conventional design in the presence of inactive shared data that would only be present in the L2 in the case of a conventional design.

### C. Configurable L2-L3 Cache

In sub-0.1$\mu$m technologies, the long access latencies of a large on-chip L2 cache [1] may be prohibitive for those applications that make use of only a small fraction of the L2 cache. Thus, for performance reasons, a three-level hierarchy with a moderate size (*e.g.*, 512KB) L2 cache will become an attractive alternative to two-level hierarchies at these feature sizes. However, the cost may be a significant increase in energy dissipation due to transfers involving the additional cache level. We demonstrate in Section V that the use of the aforementioned configurable cache structure as a replacement for conventional L2 and L3 caches can significantly reduce energy dissipation without any compromise in performance as feature sizes scale below 0.1$\mu$m.

### D. Configurable TLB Organization

Our 512-entry, fully-associative TLB can be similarly configured as shown in Figure 3 (b). There are eight TLB increments, each of which contains a CAM of 64 virtual page numbers and an associated RAM of 64 physical page numbers. Switches are inserted on the input and output buses to electrically isolate successive increments. Thus, the ability to configure a larger TLB does not degrade the access time of the minimal size (64 entry) TLB. Similar to the cache design, the TLB behaves as a two-level non-inclusive hierarchy and misses result in a second access but to the backup portion of the TLB.

## III. Dynamic Selection Mechanisms

Our configurable cache and TLB permits picking appropriate configurations and sizes based on application requirements. The different configurations spend different amounts of time and energy accessing the L1 and the lower levels of the memory hierarchy. Our heuristics improve the efficiency of the memory hierarchy by trying to minimize idle time due to memory hierarchy access. The goal is to determine the right balance between hit latency and miss rate for each application phase based on the tolerance of the phase for the hit and miss latencies. Our approach is to design the selection mechanisms to improve performance and then to introduce modifications to the heuristics that opportunistically trade off a small amount of performance for significant energy savings. These heuristics require appropriate metrics for assessing the cache/TLB performance of a given configuration during each application phase.

The two key aspects in designing a mechanism that dynamically selects one of many configurations is first, to identify when the application moves to a different phase, and second, to pick the correct configuration for that phase. A combination of profiling and compiler analysis can do an effective job identifying when the program moves in and out of a new phase, for example, a long subroutine or a loop. To minimize programmer intervention, we use hardware mechanisms to approximate the detection of phase changes - that is, we use certain hardware events to signal a phase change instead of using special instructions inserted by the compiler. We found that such a solution not only worked well, it also required very modest amounts of additional hardware. To determine the correct cache and TLB organization for a given phase, we need to estimate the effect of each organization on performance (assuming that we are trying to maximize performance). One way to do this is to simply explore the configuration space by measuring performance when executing using each configuration. While this is the simplest and most accurate way to pick the appropriate configuration, it could cause serious overheads (and result in potential inaccuracies) if this initial monitoring process lasts for a significant fraction of the duration of the phase. An alternative is to use metrics that predict the performance for each configuration so that the best configuration can be chosen without exploration [14]. This would trade some accuracy for lower overheads. As our results show later, these overheads are small enough and the phases are long enough for our range of benchmarks to make exploration the method of choice.

### A. Phase Detection

A change in program phase potentially occurs when the program enters or exits loops or subroutines. Such a change can be detected by certain hardware events – a mispredicted branch, a jump to a subroutine, I-cache misses, change in the instruction composition, *etc.* Since we are using an interval-based selection mechanism, we use the last metric to detect a phase change. We found that different phases usually have different branch frequencies – loops with differently sized loop bodies, subroutines with different average basic block sizes, *etc.* Since different phases could have the same branch frequency, but different cache behaviors, we also use the number of cache misses as an additional significant metric that indicates a phase change.

Hence, a phase change is indicated by changes in either the branch frequency or the number of cache misses.

### B. Search Heuristics

We first describe a simple interval-based scheme that uses history to pick a size for the future. Large L1 caches have a high hit rate, but also have higher access times. One of the many different L1 sizes afforded by the reconfigurable cache has the best trade-off point between the cache hit and miss times. At regular intervals (100K cycles in our simulations), the hardware counters described in Section III-A are inspected to detect a phase change and determine if a change in cache organization is required. When a change in phase is detected, each cache configuration is tried for an interval of 100K cycles. After this exploratory process, the organization that has the lowest CPI is used until the next phase change.

Our initial scheme is tuned to improve performance and thus explores the following five cache configurations: 256KB 1-way L1, 768KB 3-way L1, 1MB 4-way L1, 1.5MB 3-way L1, and 2MB 4-way L1. The 512KB 2-way L1 configuration provides no performance advantage over the 768KB 3-way L1 configuration (due to their identical access times in cycles) and thus this configuration is not used. For similar reasons, the two low-energy configurations (direct-mapped 512KB L1 and two-way set associative 1MB L1) are only used with modifications to the heuristics that reduce energy (described shortly).

At the end of each *interval* of execution (100K cycles), we examine a set of hardware counters. These hardware counters tell us the miss rate, the CPI, and the branch frequency experienced by the application in that last interval. Based on this information, the selection mechanism (which could be implemented in software or hardware) picks one of two states - stable or unstable. The former suggests that behavior in this interval is not very different from the last and we do not need to change the cache configuration, while the latter suggests that there has recently been a phase change in the program and we need to explore (or continue to explore) to determine the appropriate cache configuration.

The initial state is unstable and the initial L1 cache is chosen to be the smallest (256KB in this paper). At the end of an interval, we enter the CPI experienced for that cache size into a table. If the miss rate exceeds a certain threshold (1% in our experiments) during that interval, we switch to the next largest L1 cache configuration for the next interval of operation in an attempt to contain the working set. This exploration continues until the maximum L1 size is reached or until the miss rate is sufficiently small. At this point, the table is examined, the cache configuration with the lowest CPI is picked, the table is cleared, and we switch to the stable state. We continue to remain in the stable state while the number of misses and branches do not significantly differ from that experienced for that particular cache size during the exploration phase. When there is a change, we switch to the unstable state, return to the smallest L1 cache configuration and start exploring again. The pseudo-code for the mechanism is listed below.

```
Initializations and definitions:
  base_br_noise = 4500; base_miss_rate_noise = 450;
  br_incr = 1000; br_decr = 50;
  miss_rate_incr = 100; miss_rate_decr = 5;
  miss_rate_noise = base_miss_rate_noise;
  br_noise = base_br_noise;
  state = UNSTABLE;

Repeat the following every 100K cycles:
(inputs: num_miss, num_br, CPI)
if (state == STABLE)
    if ((|num_miss-last_num_miss|) < miss_rate_noise &&
        (|num_br-last_num_br|) < br_noise)
        miss_rate_noise = max(miss_rate_noise-miss_rate_decr,
                              base_miss_rate_noise);
        br_noise = max(br_noise - br_decr, base_br_noise);
    else
        last_cache_size = cache_size;
        cache_size = SMALLEST; state = UNSTABLE;

else if (state == UNSTABLE)
    record CPI, num_miss, num_br;
    if ((num_miss > THRESHOLD) && (cache_size != MAX))
        Increase cache_size;
    else
        cache_size = that with best CPI; state = STABLE;
        last_num_miss = num_miss recorded for selected size;
        last_num_br = num_br recorded for selected size;
        if (cache_size == last_cache_size)
          miss_rate_noise= miss_rate_noise + miss_rate_incr;
          br_noise = br_noise + br_incr;
```

Different applications see different variations in the number of misses and branches as they move across application phases. Hence, instead of using a single fixed number as the threshold to detect phase changes, we vary it dynamically. If an exploration results in picking the same cache size as before, the noise thresholds are increased to discourage such needless explorations. Likewise, every interval spent in the stable state causes a slight decrement in the noise thresholds in case they had been set too high.

The miss rate threshold ensures that we explore larger cache sizes only if required. Note that a high miss rate need not necessarily have a large impact on performance because of the ability of dynamic superscalar processors to hide L2 latencies.

Clearly, such an interval-based mechanism is best suited to programs that can sustain uniform behavior for a number of intervals. While switching to an unstable state, we also move to the smallest L1 cache configuration as a form of "damage control" for programs that have irregular behavior. This choice ensures that for these programs, more time is spent at the smaller cache sizes and hence performance is similar to that using a conventional cache hierarchy. We also keep track of how many intervals are spent in stable and unstable states. For every interval in stable state, a saturating counter is incremented by one and is decremented by four for every interval in unstable state. If the counter value goes below a threshold (-25 in our simulations), we conclude that we are spending too much time exploring, *i.e.*, the program behavior is not suited to an interval-based scheme and we remain fixed at the smallest sized cache.

Mis-maps, as described in Section II-B, occur when the cache uses four subarrays soon after having used two subar-

rays (or vice versa). This usually happens only once during each exploration phase. Given the infrequent occurrence of explorations for most programs, we found that the number of mis-maps are usually few and do not impact performance by much. Among our benchmarks, *health* has the highest number of mis-maps, encountering one per 1000 committed instructions. The average increase in memory energy consumption due to the need to examine twice as many tags on an L2 look-up is 0.7%.

We envision that the selection algorithm would be implemented in software. Every 100K cycles, a low-overhead software handler is invoked that examines the hardware counters and updates the state as necessary. This imposes minimal hardware overhead and allows flexibility in terms of modifying the selection algorithm. We estimate the code size of the handler to be about 120 static assembly instructions, only a fraction of which is executed during each invocation, resulting in a net overhead of less than 0.1%. In terms of hardware cost, we need 20-bit counters for the number of misses, loads, cycles, instructions, and branches, in addition to a state register. This amounts to less than 8,000 transistors.

### C. Performance Metrics

Cache miss rates provide a first order approximation of the cache requirements of an application, but they do not directly reflect the effects of various cache sizes on memory stall cycles. We present and evaluate a metric that quantifies this effect in order to better identify the appropriate cache configuration.

The actual number of memory stall cycles is a function of the time taken to satisfy each cache access and the ability of the out-of-order execution window to overlap other useful work while these accesses are made. Load latency tolerance has been characterized in [33], and [16] introduces two hardware mechanisms for estimating the criticality of a load. One of these monitors the issue rate while a load is outstanding and the other keeps track of the number of instructions dependent on that load. While these schemes are easy to implement, they are not very accurate in capturing the number of stall cycles resulting from an outstanding load. We propose an approach that more accurately characterizes load stall time and further breaks this down as stalls due to cache hits and misses. The goal is to provide insight to the selection algorithm as to whether it is necessary to move to a larger or smaller L1 cache configuration (or not to move at all) for each application phase.

To every entry in the register map table, we add one bit that indicates whether the given (logical) register is to be written by a load instruction. In addition, for every entry in the Register Update Unit (RUU[2]), we add one bit per operand that specifies if the operand is produced by a load (which can be deduced from the additional register map table bits) and another specifying if the load was a hit (the initial value upon insertion into the RUU) or a miss. Each cycle, every instruction in the RUU that directly depends on a load increments one of two global *intolerance counters* if (i) all operands except for the operand produced by a load are ready, (ii) a functional unit is available, and (iii) there are free issue slots in that cycle. For every cycle that these conditions are met up to the point that the load-dependent instruction issues, the *hit intolerance counter* is incremented unless a cache miss is detected for the load that it is dependent on; if such a miss occurs, the hit/miss bit is switched and the *miss intolerance counter* is incremented each cycle that the above three conditions are met until the point at which the instruction issues. If more than one operand of an instruction is produced by a load, a heuristic is used to choose the hit/miss bit of one of the operands. In our simulations, we choose the operand corresponding to the load that issued first. This scheme requires only very minor changes to existing processor structures and two additional performance counters, while providing a fairly accurate assessment of the relative impact of the hit time and the miss time of the current cache configuration on actual execution time of a given program phase.

[2]The RUU [31] is a unified queue and reorder buffer that holds all instructions that have dispatched and not committed.

### D. Reconfiguration on a Per-Subroutine Basis

As previously mentioned, the interval-based scheme will work well only if the program can sustain its execution phase for a number of intervals. This limitation may be overcome by collecting statistics and making subsequent configuration changes on a *per-subroutine* basis. The finite state machine that was used for the interval-based scheme is now employed for each subroutine – *i.e.*, the event used to determine the phase change is not the instruction composition, but the use of a 'jump subroutine' or a 'return' instruction. This requires maintaining a table with CPI values at different cache sizes and the next size to be picked for a limited number of subroutines (100 in this paper). To focus on the important routines, we only monitor those subroutines whose invocations exceed a certain threshold of dynamic instructions (1000 in this paper). When a subroutine is invoked, its table is looked up and a change in cache configuration is effected depending on the table entry for that subroutine. When a subroutine exits, it updates the table based on the statistics collected during that invocation. A stack is used to checkpoint counters on every subroutine call so that statistics are maintained for each subroutine invocation.

We investigated two subroutine-based schemes. In the *non-nested* approach, statistics are collected for a subroutine and its callees. Cache size decisions for a subroutine are based on these statistics collected for the call-graph rooted at this subroutine. Once the cache configuration is changed for a subroutine, none of its callees can change the configuration unless the outer subroutine returns. Thus, the callees inherit the size of their callers because their statistics play a role in determining the configuration of the caller. In the *nested* scheme, each subroutine collects statistics for the period when it is the top of the subroutine call stack. Thus, every subroutine invocation is looked upon as a possible change in phase.

These schemes work well only if successive invocations

of a particular subroutine are consistent in their behavior. A common case where this is not true is that of a recursive program. We handle this situation by not letting a subroutine update the table if there is an outer invocation of the same subroutine, *i.e.*, we assume that only the outermost invocation is representative of the subroutine and that successive outermost invocations will be consistent in their behavior.

If the stack used to checkpoint statistics overflows, we assume that future invocations will inherit the size of their caller for the non-nested case and will use the minimum sized cache for the nested case. While the stack is in a state of overflow, subroutines will be unable to update the table. If a table entry is not found while entering a subroutine, the default smallest sized cache is used for that subroutine for the nested case. Since the simpler non-nested approach generally outperformed the nested scheme, we only report results for the former in Section V.

### *E. TLB Reconfiguration*

In addition to cache reconfiguration, we also progressively change the TLB configuration on an interval-by-interval basis. A counter tracks TLB miss handler cycles and the L1 TLB size is increased if this counter exceeds a threshold (3% in this paper) of the total execution time counter for an interval. A single bit is added to each TLB entry that is set to indicate if it has been used in an interval (and is flash cleared in hardware at the start of an interval). At the end of each interval, the number of TLB entries that have their bit set is counted. This can be done in hardware with fairly simple and energy-efficient logic. Similar logic that aggregates usage information within the issue queue has been proposed by Buyuktosunoglu et al [11]. The L1 TLB size is decreased if the TLB usage is less than half.

For the cache reconfiguration, we chose an interval size of 100K cycles so as to react quickly to changes without letting the selection mechanism pose a high cycle overhead. For the TLB reconfiguration, we used a one million cycle interval so that an accurate estimate of TLB usage could be obtained. A smaller interval size could result in a high TLB miss rate and a low TLB usage over the same interval.

### *F. Energy-Aware Modifications*

There are two energy-aware modifications to the selection mechanisms that we consider. The first takes advantage of the inherently low-energy configurations (those with direct-mapped 512KB and two-way set associative 1MB L1 caches). With this approach, the selection mechanism simply uses these configurations in place of the 768KB 3-way L1 and 1MB 4-way L1 configurations.

A second approach is to serially access the tag and data arrays of the L1 data cache. Conventional L1 caches always perform parallel tag and data look-up to reduce hit time, thereby reading data out of multiple cache ways and ultimately discarding data from all but one way. By performing tag and data look-up in series, only the data way associated with the matching tag can be accessed, thereby reducing energy consumption. Hence, our second low-energy mode operates just like the interval-based scheme as before, but accesses the set-associative cache configurations by serially reading the tag and data arrays.

TABLE I
ARCHITECTURAL PARAMETERS.

| | |
|---|---|
| Branch predictor | comb. of bimodal & 2-level gshare;<br>Combining pred. entries - 1024;<br>bimodal - 2048 entries<br>Gshare - level1/2 - 1024/4096<br>RAS entries - 32; BTB - 2048 sets |
| Branch mispred. latency | 8 cycles |
| Fetch, decode, issue width | 4 |
| RUU and LSQ entries | 64 and 32 |
| L1 I-cache | 2-way; 64KB (0.1$\mu$m),<br>32KB (0.035$\mu$m) |
| Memory latency | 80 cycles (0.1$\mu$m),<br>114 cycles (0.035$\mu$m) |
| Integer ALUs/mult-div | 4/2 |
| FP ALUs/mult-div | 2/1 |

### *G. L2/L3 Reconfiguration*

The selection mechanism for the L2/L3 reconfiguration is similar to the simple interval-based mechanism for the L1/L2. In addition, because we assume that the L2 and L3 caches (both conventional and configurable) already use serial tag/data access to reduce energy dissipation, the energy-aware modifications would provide no additional benefit for L2/L3 reconfiguration. (Recall that performing the tag look-up first makes it possible to turn on only the required data way within a subarray, as a result of which, all configurations consume the same amount of energy for the data array access.) Finally, we did not simultaneously examine TLB reconfiguration so as not to vary the access time of the fixed L1 data cache. The motivation for these simplifications was due to our expectation that dynamic L2/L3 cache configuration would yield mostly energy saving benefits, due to the fact that we were not altering the L1 cache configuration (the organization of which has the largest memory performance impact for most applications). To further improve our energy savings at minimal performance penalty, we also modified the search mechanism to pick a larger sized cache if it performed almost (within 95%) as well as the best performing cache during the exploration, thus reducing the number of transfers between the L2 and L3.

## IV. EVALUATION METHODOLOGY

### *A. Simulation Methodology*

We used Simplescalar-3.0 [10] for the Alpha AXP ISA to simulate an aggressive 4-way superscalar out-of-order processor. The architectural parameters used in the simulation are summarized in Table I.

The data memory hierarchy is modeled in great detail. For example, contention for all caches and buses in the memory hierarchy as well as for writeback buffers is modeled. The line size of 128 bytes was chosen because it yielded a much lower miss rate for our benchmark set than smaller line sizes.

TABLE II
BENCHMARKS.

| Benchmark | Suite | Datasets | Simulation window (instrs) | 64KB-2way L1 miss rate | % of instrs that are loads |
|---|---|---|---|---|---|
| em3d | Olden | 20,000 nodes, arity 20 | 1000M-1100M | 20% | 36% |
| health | Olden | 4 levels, 1000 iters | 80M-140M | 16% | 54% |
| mst | Olden | 256 nodes | entire program 14M | 8% | 18% |
| compress | SPEC95 INT | ref | 1900M-2100M | 13% | 22% |
| hydro2d | SPEC95 FP | ref | 2000M-2135M | 4% | 28% |
| apsi | SPEC95 FP | ref | 2200M-2400M | 6% | 23% |
| swim | SPEC2000 FP | ref | 2500M-2782M | 10% | 25% |
| art | SPEC2000 FP | ref | 300M-1300M | 16% | 32% |

For both configurable and conventional TLB hierarchies, a TLB miss at the first level results in a look-up in the second level. A miss in the second level results in a call to a TLB handler that is assumed to complete in 30 cycles. The page size is 8KB.

### B. Benchmarks

We used a mix of programs from SPEC95, SPEC2000, and the Olden suite [29]. These programs were chosen because they have high miss rates for the L1 caches we considered. For programs with low miss rates for the smallest cache size, the dynamic scheme affords no advantage and behaves like a conventional cache. The benchmarks were compiled with the Compaq cc, f77, and f90 compilers at an optimization level of O3. Warmup times were determined for each benchmark, and the simulation was fast-forwarded through these phases. A further million instructions were simulated in detail to prime all structures before starting the performance measurements. The window size was chosen to be large enough to accommodate at least one outermost iteration of the program, where applicable. Table II summarizes the benchmarks and their memory reference properties (the L1 miss rate and load frequency).

### C. Timing and Energy Estimation

We investigated two future technology feature sizes: 0.1 and 0.035$\mu$m. For the 0.035$\mu$m design point, we use the cache latency values of Agarwal et al. [1] whose model parameters are based on projections from the Semiconductor Industry Association Technology Roadmap [4]. For the 0.1$\mu$m design point, we use the cache and TLB timing model developed by McFarland [25] to estimate timings for both the configurable and conventional caches and TLBs. McFarland's model contains several optimizations, including the automatic sizing of gates according to loading characteristics, and the careful consideration of the effects of technology scaling down to 0.1$\mu$m technology [26]. The model integrates a fully-associative TLB with the cache to account for cases in which the TLB dominates the L1 cache access path. This occurs, for example, for all of the conventional caches that were modeled as well as for the minimum size L1 cache (direct mapped 256KB) in the configurable organization.

For the global wordline, local wordline, and output driver select wires, we recalculate cache and TLB wire delays using RC delay equations for repeater insertion [13]. Repeaters are used in the configurable cache as well as in the conventional L1 cache whenever they reduce wire propagation delay.

TABLE III
SIMULATED L1/L2 CONFIGURATIONS.

| | |
|---|---|
| A | Base excl. cache; 256KB 1-way L1 & 1.75MB 14-way L2 |
| B | Base incl. cache; 256KB 1-way L1 & 2MB 16-way L2 |
| C | Base incl. cache; 64KB 2-way L1 & 2MB 16-way L2 |
| D | Interval-based dynamic scheme |
| E | Subroutine-based non-nested scheme |
| F | Interval-based with energy-aware configurations |
| G | Interval-based with serial tag and data access |

We estimate cache and TLB energy dissipation using a modified version of the analytical model of Kamble and Ghose [21]. This model calculates cache energy dissipation using similar technology and layout parameters as those used by the timing model (including voltages and all electrical parameters appropriately scaled for 0.1$\mu$m technology). The TLB energy model was derived from this model and included CAM match line precharging and discharging, CAM wordline and bitline energy dissipation, as well as the energy of the RAM portion of the TLB. For main memory, we include only the energy dissipated due to driving the off-chip capacitive busses.

For all L2 and L3 caches (both configurable and conventional), we assume serial tag and data access and selection of only one of 16 data banks at each access, similar to the energy-saving approach used in the Alpha 21164 L2 cache [9]. The conventional L1 caches were divided into two subarrays, only one of which is selected at each access. Thus, the conventional cache hierarchy against which we compared our reconfigurable hierarchy was highly optimized for fast access time and low energy dissipation.

Detailed event counts were captured during the simulations of each benchmark. These event counts include all cache and TLB operations and are used to obtain final energy estimations.

### D. Simulated Configurations

Table III shows the conventional and dynamic L1/L2 schemes that were simulated. We compare our dynamic schemes with three conventional configurations that are identical in all respects, except the data cache hierarchy. The first uses a two-level non-inclusive cache, with a direct mapped 256KB L1 cache backed by a 14-way 1.75MB L2

cache (configuration A). The L2 associativity results from the fact that 14 ways remain in each 512KB structure after two of the ways are allocated to the 256KB L1 (only one of which is selected on each access). Comparison of this scheme with the configurable approach demonstrates the advantage of resizing the first level. We also compare with a two-level inclusive cache that consists of a 256KB direct mapped L1 backed by a 16-way 2MB L2 (configuration B). This configuration serves to measure the impact of the non-inclusive policy of the first base case on performance (a non-inclusive cache performs worse because every miss results in a swap or writeback, which causes greater bus and memory port contention.) We also compare with a 64KB 2-way inclusive L1 and 2MB of 16-way L2 (configuration C), which represents a typical configuration in a modern processor and ensures that the performance gains for our dynamically sized cache are not obtained simply by moving from a direct mapped to a set associative cache. For both the conventional and configurable L2 caches, the access time is 15 cycles due to serial tag and data access and bus transfer time, but is pipelined with a new request beginning every four cycles. The conventional TLB is a two-level inclusive TLB with 64 entries in the first level and 448 entries in the second level with a 6 cycle look-up time.

For L2/L3 reconfiguration, we compare our interval-based configurable cache with a conventional three-level on-chip hierarchy. In both, the L1 data and instruction caches are 32KB two-way set associative with a three cycle latency, reflecting the smaller L1 caches and increased latency likely required at 0.035$\mu$m geometries [1]. For the conventional hierarchy, the L2 cache is 512KB two-way set associative with a 21 cycle latency and the L3 cache is 2MB 16-way set associative with a 60 cycle latency. Serial tag and data access is used for both L2 and L3 caches to reduce energy dissipation (the 21 and 60 cycle latencies take this serialization into account).

## V. Results

### A. L1/L2 Performance Results

The performance afforded by a given cache organization is determined greatly by the L1 miss rate and to a lesser extent by the L1 access time. A number of programs have working sets that do not fit in today's L1 caches. For our chosen memory-intensive benchmark set, half of the total execution time can be attributed to memory hierarchy accesses (as shown by later graphs). Increasing the size of the L1 and thereby reducing the miss rate has a big impact on CPI for such programs. At the same time, the increased access time for the L1 results in poorer performance for other non-memory-intensive programs. For example, we observed that for most SPEC95 integer programs, each additional cycle in the L1 access time resulted in a 4-5% performance loss.

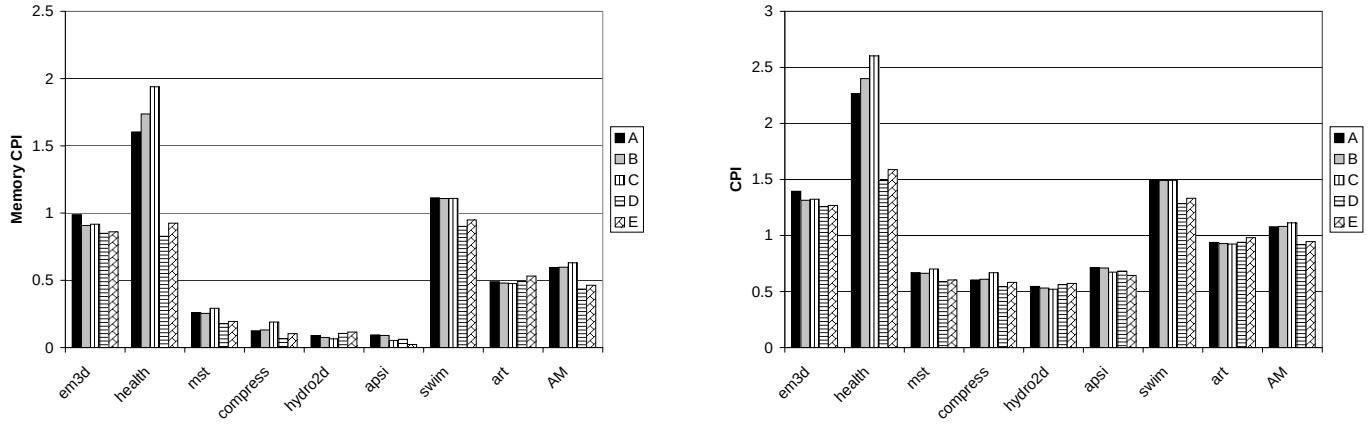
The reconfigurable L1/L2 cache provides a number of attractive design points for both memory-intensive and non-memory-intensive applications. Programs that do not have large working sets and do not suffer from many conflict misses can use the smaller and faster 256KB direct-mapped L1. Programs with large working set sizes, whose execution times are dominated by accesses to the L2 and beyond can use large L1 sizes so that most accesses are satisfied by a single cache look-up. While each access now takes longer, its performance effect is usually smaller than the cost of a higher miss rate. Moving to a larger cache size not only handles many of the capacity misses, it also takes care of a number of conflict misses as the associativity is increased in tandem. In our experiments, the combined L1-L2 hierarchy has a 2MB capacity. If the working set of the program is close to 2MB, the entire cache can be used as the L1. This not only reduces the miss rate, it also eliminates the L2 look-up altogether, reducing the effective memory access time. Our benchmark set represents programs with various working set sizes and associativity needs (even for different phases of the same program) and the dynamic selection mechanisms adapt the underlying L1-L2 cache hierarchy to these needs. A couple of the programs also have frequently changing needs that cannot be handled by our simple interval-based scheme.

TABLE IV
Contribution of the cache and the TLB to speedup or slowdown in the dynamic scheme and the number of explorations.

| | Cache contribution | TLB contribution | Cache expl | TLB changes |
|---|---|---|---|---|
| em3d | 73% | 27% | 10 | 2 |
| health | 33% | 67% | 27 | 2 |
| mst | 100% | 0% | 5 | 3 |
| compress | 64% | 36% | 54 | 2 |
| hydro2d | 100% | 0% | 19 | 0 |
| apsi | 100% | 0% | 63 | 27 |
| swim | 49% | 51% | 5 | 6 |
| art | 100% | 0% | 11 | 5 |

Figure 5 shows the memory CPI and total CPI achieved by the conventional and configurable interval and subroutine-based schemes for the various benchmarks. The memory CPI is calculated by subtracting the CPI achieved with a simulated system with a perfect cache (all hits and one cycle latency) from the CPI with the realistic memory hierarchy. In comparing the arithmetic mean (AM) of the memory CPI performance, the interval-based configurable scheme outperforms the best-performing conventional scheme (B) (measured in terms of a percentage reduction in memory CPI) by 27%, with roughly equal cache and TLB contributions as is shown in Table IV. For each application, this table also presents the number of cache and TLB explorations that resulted in the selection of different sizes. In terms of overall performance, the interval-based scheme achieves a 15% reduction in CPI. The benchmarks with the biggest memory CPI reductions are *health* (52%), *compress* (50%), *apsi* (31%), and *mst* (30%).

The dramatic improvements with *health* and *compress* are due to the fact that particular phases of these applications perform best with a large L1 cache even with the resulting higher hit latencies (for which there is reasonably high tolerance within these applications). For *health*,




(a) Memory CPI results. (b) CPI results.

Fig. 5. Performance for conventional (A, B, and C), interval-based (D), and subroutine-based (E) configurable schemes. Memory CPI is shown in (a) and CPI in (b).

the configurable scheme settles at the 1.5MB cache size for most of the simulated execution period, while the 768KB configuration is chosen for much of *compress*' execution period. Note that TLB reconfiguration also plays a major role in the performance improvements achieved. These two programs best illustrate the mismatch that often occurs between the memory hierarchy requirements of particular application phases and the organization of a conventional memory hierarchy, and how an intelligently-managed configurable hierarchy can better match on-chip cache and TLB resources to these execution phases. Note that while some applications stay with a single cache and TLB configuration for most of their execution window, others demonstrate the need to adapt to the requirements of different phases in each program (see Table IV). Regardless, the dynamic schemes are able to determine the best cache and TLB configurations, which span the entire range of possibilities, for each application during execution.

For two of the programs, *em3d* and *swim*, the heuristics choose the entire 2MB cache space as the L1 for most of their execution time. These are examples of programs that have working sets larger than 2MB. Performance improvements are seen not just because of the lower miss rate afforded by a large L1, but also by eliminating the 15-cycle L2 look-up altogether.

The results for *art* and *hydro2d* demonstrate how the dynamic reconfiguration may in some cases degrade performance. These applications are very unstable in their behavior and do not remain in any one phase for more than a few intervals. *Art* also does not fit in 2MB, so there is no size that causes a sufficiently large drop in CPI to merit the cost of exploration. However, the dynamic scheme identifies that the application is spending more time exploring than in stable state and turns exploration off altogether. Since this happens early enough in the case of *art* (the simulation window is also much larger), *art* shows no overall performance degradation, while *hydro2d* has a 3% slowdown. This result illustrates that compiler analysis to identify such "unstable" applications and override the dynamic selection mechanism with a statically-chosen cache configuration may be beneficial.

In terms of the effect of TLB reconfiguration, *health, swim,* and *compress* benefit the most from using a larger TLB. *Health* and *compress* perform best with 256 and 128 entries, respectively, and the dynamic scheme settles at these sizes. *Swim* shows phase change behavior with respect to TLB usage, resulting in five stable phases requiring either 256 or 512 TLB entries.

### A.1 Reconfiguration on a Per-Subroutine Basis

Figure 5 also allows us to compare the interval and subroutine-based schemes. As the results show, the simpler interval-based scheme usually outperforms the subroutine-based approach. If the application phase behavior is data or time-dependent rather than code location dependent, the subroutine-based scheme will be slower to adapt to the change. In addition, there is potential for instability across subroutine invocations especially if the same procedure is called from multiple locations or phases in the program. The exception in our benchmark suite is *apsi*, for which the subroutine-based scheme improves performance relative to the interval-based approach as each subroutine exhibits consistent behavior across subroutine invocations. With the interval-based scheme, *apsi* shows inconsistent behavior across intervals (indicated by the large number of explorations in Table IV), causing it to thrash between a 256KB L1 and a 768KB L1. However, the interval-based scheme is better able to capture application behavior on average than the subroutine-based scheme, in addition to being more practical since it requires simpler hardware. Hence, we use the interval-based scheme as the basis for the rest of our analysis.

### A.2 Limiting the Exploration Process

The exploration process that is invoked for a few intervals on every phase change accounts for a small portion of the total run-time. We limited the exploration process by not evaluating larger cache configurations if the L1 miss rate was less than 1%. By removing this constraint and causing all cache configurations to be evaluated on every

TABLE V
Latencies (in cycles) for the various cache organizations at the three different processor frequency points (TLB size is 64 in each case).

| Cache organization | Latency at 1GHz | Latency at 1.5GHz | Latency at 2GHz |
|---|---|---|---|
| 256KB 1-way | 2.0 | 3.0 | 4.0 |
| 512KB 2-way | 2.5 | 4.0 | 5.0 |
| 768KB 3-way | 2.5 | 4.0 | 5.0 |
| 1MB 4-way | 3.0 | 4.0 | 6.0 |
| 1.5MB 3-way | 4.0 | 6.0 | 8.0 |
| 2MB 4-way | 4.5 | 7.0 | 9.0 |

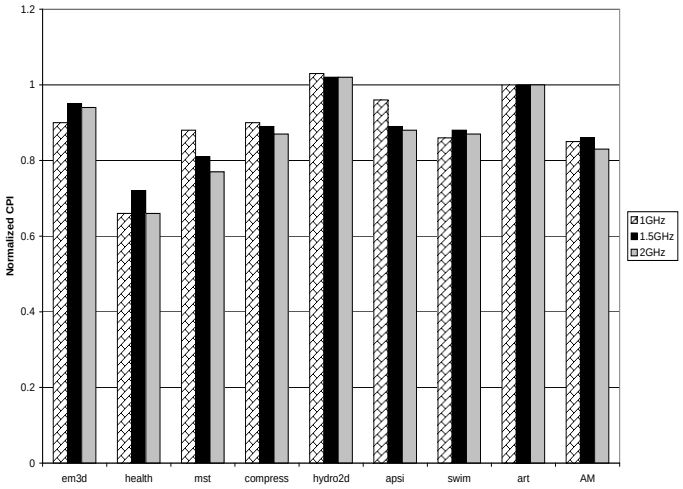



Fig. 6. Normalized CPIs with the dynamic interval-based scheme assuming processor clock speeds of 1, 1.5, and 2GHz. The CPIs have each been normalized to their respective base cases.

phase change, we saw only a 1% overall slowdown in CPI.

An L1 miss rate of less than 1% indicates that the program is not limited by L2 accesses. However, a program may have a higher L1 miss rate and yet, not be limited by the L2 accesses if the program has the ability to tolerate these longer load latencies. Hence, to evaluate if more accurate metrics could help limit the exploration process further, we used miss intolerance to quantify the effect of L1 misses on the program. Miss intolerance indicates the number of cycles that a program's completion is delayed because of accesses to the L2 and beyond, and is described in Section III-C. Larger cache configurations were explored only if the L1 miss intolerance exceeded a certain threshold. Only *apsi* showed a 2% CPI improvement, with no overall CPI improvement being registered.

Overall, limiting the exploration process by using L1 miss rates as the metric improves performance by 1%, and using more complicated metrics does not improve this performance by much more, indicating that the exploration process is not a large overhead for an interval size of 100,000 cycles.

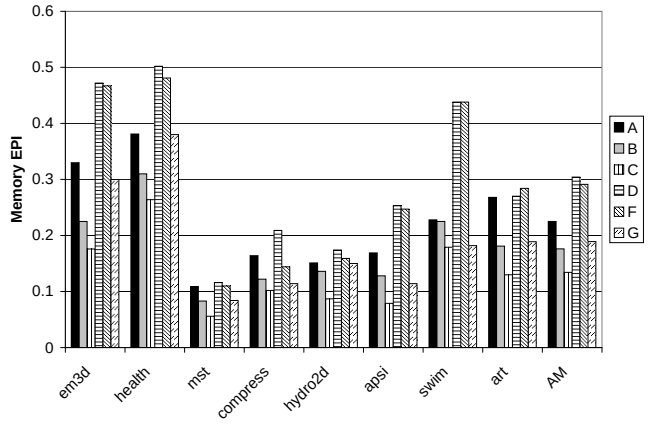
### A.3 Sensitivity to Processor Clock Speed

The above results demonstrate potential performance improvement for one technology point and microarchitecture. In order to determine the sensitivity of our qualitative results to different technology points and microarchitectural trade-offs, we varied the processor pipeline speed relative to the memory latencies (keeping the memory hierarchy delays in ns fixed). Apart from the 1GHz processor clock that has been used for $0.1\mu$ technology throughout this paper, we also evaluated the effect of 1.5GHz and 2GHz processor clocks. Table V summarizes the delays of the various cache organizations at each frequency. For 1.5GHz and 2GHz processors, the half-cycle latencies are eliminated, thereby simplifying the design. Figure 6 shows the normalized CPIs because of the dynamic interval-based scheme for each frequency point (the CPIs have been normalized to their respective base cases, so a CPI of 0.9 indicates a 10% improvement). The overall improvements for the three cases are 15%, 14%, and 17%, demonstrating that our results are valid for various processor-memory speed ratios.

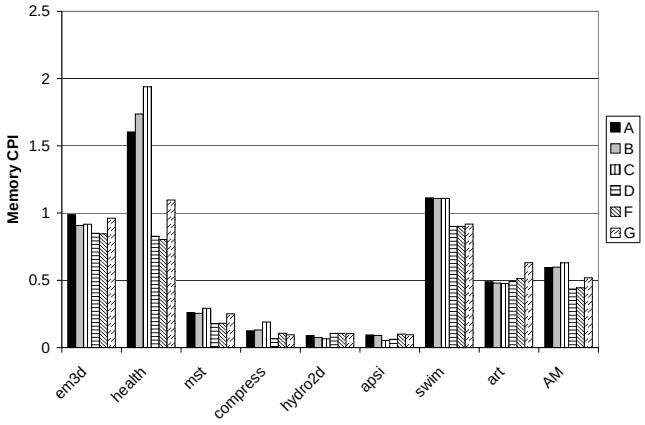
### B. Energy-Aware Configuration Results

In this subsection, we focus on the energy consumption of the on-chip memory hierarchy. The memory energy per instruction (memory EPI, with each energy unit measured in nanoJoules) results of Figure 7 (a) illustrate how as is usually the case with performance optimizations, the cost of the performance improvement due to the configurable scheme is a significant increase in energy dissipation. This is caused by the fact that energy consumption is proportional to the associativity of the cache and our configurable L1 uses larger set-associative caches. For this reason, we explore how the energy-aware improvements may be used to provide a more modest performance improvement yet with a significant reduction in memory EPI relative to a pure performance approach.

From Figure 7 (a), we observe that merely selecting the energy-aware cache configurations (scheme F) has only a nominal impact on energy. In contrast, operating the L1 cache in a serial tag and data access mode (G) reduces memory EPI by 38% relative to the baseline interval-based scheme (D), bringing it in line with the best overall-performing conventional approach (B). For *compress* and *swim*, this approach even achieves roughly the same energy, with significantly better performance (see Figure 7 (b)), than conventional configuration C, whose 64KB two-way L1 data cache activates half the amount of cache every cycle than the smallest L1 configuration (256KB) of the configurable schemes. In addition, because the selection scheme automatically adjusts for the higher hit latency of serial access, this energy-aware configurable approach reduces memory CPI by 13% relative to the best-performing conventional scheme (B). Figure 8 shows memory energy-delay product for the various configurable schemes. In comparison to the conventional schemes A and B with the same total amount of cache and TLB, our serial tag-data approach (G) results in a 25% and 8% improvement in the memory energy-delay product respectively. The energy-delay product of the interval-based scheme (D) is comparable to that of base case A. Thus, the energy-aware approach may be used to provide more balanced improve-




(a) Memory EPI results.




(b) Memory CPI results.

Fig. 7. Results for conventional (A, B, and C), interval-based (D), and energy-aware (F and G) configurable schemes. Memory EPI (in nanoJoules) is shown in (a) and memory-CPI in (b).

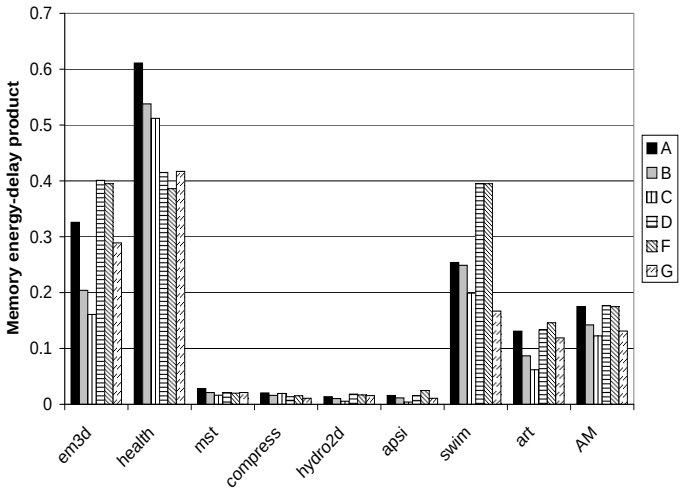



Fig. 8. Memory energy-delay product for conventional (A, B, and C), interval-based (D), and energy-aware (F and G) configurable schemes.

ments in both performance and energy in portable applications where design constraints such as battery life are of utmost importance. Furthermore, as with the dynamic voltage and frequency scaling approaches used today, this mode may be switched on under particular environmental conditions (*e.g.*, when remaining battery life drops below a given threshold), thereby providing on-demand energy-efficient operation.

### *C. L2/L3 Performance and Energy Results*

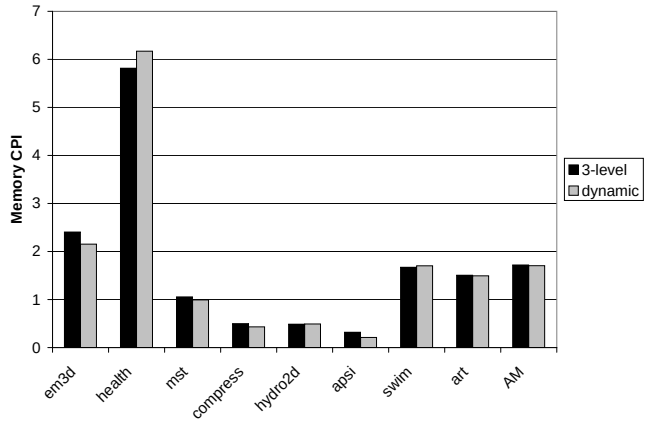
While L1 reconfiguration improves performance, it may consume more energy than conventional approaches if higher L1 associative configurations are enabled. To reduce energy, mechanisms such as serial tag and data access (as described in the previous subsection) have to be used. Since L2 and L3 caches are often already designed for serial tag and data access to save energy, reconfiguration at these lower levels of the hierarchy would not increase the energy consumed. Instead, they stand to decrease it by reducing the number of data transfers that need to be done between the various levels, *i.e.*, by improving the efficiency of the memory hierarchy.

Thus, we investigate the energy benefits of providing a configurable L2/L3 cache hierarchy with a fixed L1 cache as on-chip cache delays significantly increase with sub-0.1$\mu$m geometries. Due to the prohibitively long latencies of large caches at these geometries, a three-level cache hierarchy becomes an attractive design option from a performance perspective. We use the parameters from Agarwal et al. [1] for 0.035$\mu$m technology to illustrate how dynamic L2/L3 cache configuration can match the performance of a conventional three-level hierarchy while dramatically reducing energy dissipation.

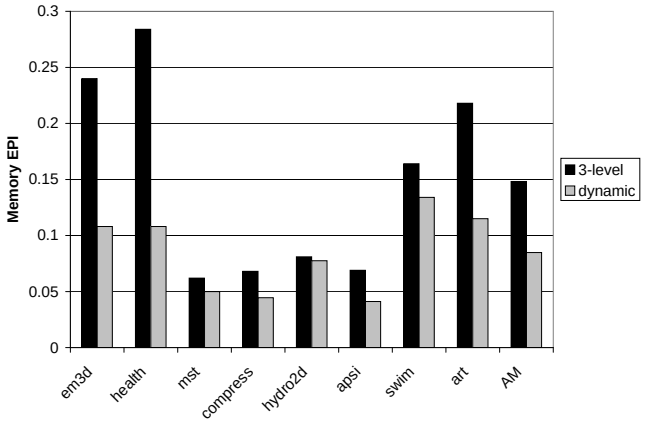
Figure 9 compares the performance and energy of the conventional three-level cache hierarchy with the configurable scheme (Recall that TLB configuration was not attempted so the improvements are completely attributable to the cache.). Since the L1 cache organization has the largest impact on cache hierarchy performance, as expected, there is little performance difference between the two, as each uses an identical conventional L1 cache. However, the ability of the dynamic scheme to adapt the L2/L3 configuration to the application results in a 42% reduction in memory EPI on average. The savings are caused by the ability of the dynamic scheme to use a larger L2, and thereby reduce the number of transfers between L2 and L3. Having only a two-level cache would, of course, eliminate these transfers altogether, but would be detrimental to program performance because of the large 60-cycle L2 access. Thus, in contrast to this approach of simply opting for a lower energy, and lower performing, solution (the two-level hierarchy), dynamic L2/L3 cache configuration can improve performance while dramatically improving energy efficiency.

## VI. Related Work

In order to address the growing gap between memory and processor speeds, techniques such as non-blocking caches [15] and hardware and software-based prefetching [20], [27] have been proposed to reduce memory latency. However, their effectiveness can be greatly improved by changing the




(a) Memory CPI results.




(b) Memory EPI results.

Fig. 9. Results for conventional three-level and dynamic cache hierarchies. Memory CPI is shown in (a) and memory EPI (in nanoJoules) in (b).

underlying structure of the memory hierarchy.

Recently, Ranganathan, Adve, and Jouppi [28] proposed a reconfigurable cache in which a portion of the cache could be used for another function, such as an instruction reuse buffer. Although the authors show that such an approach only modestly increases cache access time, fundamental changes to the cache may be required so that it may be used for other functionality as well, and long wire delays may be incurred in sourcing and sinking data from potentially several pipeline stages.

Dahlgren and Stenstrom [12] describe a cache whose organization can be changed by the compiler on a per-application basis. To handle conflict misses in a direct-mapped cache, they break the cache into multiple subunits and map different virtual address regions to these different subunits. This changes the way the cache is indexed. They also propose using different cache line sizes for different address ranges. Veidenbaum et al [34] also talk about such a reconfigurable cache, where the cache line size can be changed dynamically based on the spatial locality exhibited by the program. These changes are not done at the layout level – the cache has a small line size and depending on the program needs, an appropriate number of adjacent cache lines are fetched on a miss.

Albonesi [3] proposed the disabling of data cache ways for programs with small working sets to reduce energy consumption. A similar proposal by Yang et al [35] that reduces the number of sets in an instruction cache helps reduce leakage power for programs with small instruction working sets. In our approach, parts of the cache are never turned off – their allocations between the L1 and L2 are changed.

In an attempt to reduce the TLB miss rate, Romer et al [30] proposed the use of superpages. Contiguous virtual addresses that are accessed simultaneously are brought together to form a larger superpage. This requires the copy of the different physical pages into contiguous physical locations. In this way, a single TLB entry can then be used to translate a much larger set of virtual addresses. While this proves to be a very effective way to reduce TLB miss rates, it imposes some time and space overheads to copy the pages and to monitor and identify candidate pages for superpage promotion. In our design, the size of the TLB is increased to reduce the TLB miss rate, and an occasional IPC penalty is paid because of a potentially larger access time.

Various works [16], [32], [33] have characterized load latency tolerance and metrics for identifying critical loads. Such metrics could prove useful in determining the cache requirements for a program phase (tolerance to a longer hit latency, tolerance to cache misses, etc), but we found that such hints do not improve the performance of the selection mechanisms.

## VII. Conclusions

We have described a novel configurable cache and TLB as a higher performance and lower energy alternative to conventional on-chip memory hierarchies. Cache and TLB reconfiguration is effected by leveraging repeater insertion to allow dynamic speed/size trade-offs while limiting the impact of speed changes to within the memory hierarchy. Our results demonstrate that a simple interval-based configuration management algorithm is sufficient to achieve good performance. The algorithm is able to dynamically balance the trade-off between an application's hit and miss intolerance using CPI as the ultimate metric to determine appropriate cache size and speed. At 0.1$\mu$m technologies, our results show an average 15% reduction in CPI in comparison with the best conventional L1-L2 design of comparable total size, with the benefit almost equally attributable on average to the configurable cache and TLB. Furthermore, energy-aware enhancements to the algorithm trade off a more modest performance improvement for a significant reduction in energy. These results are not qualitatively affected by changes in relative processor speeds. Projecting to a 3-level cache hierarchy potentially necessitated by sub-micron technologies, we show an average 42% reduction in memory hierarchy energy at 0.035$\mu$m technology when compared to a conventional design.

Future work includes investigating the use of compiler

support for applications where an interval-based scheme is unable to effectively capture the phase changes in an application. Compiler support would be beneficial both to select appropriate adaptation points as well as to predict an application's working set sizes and correspondingly, the appropriate cache sizes. Finally, improvements at the circuit and microarchitectural levels can be pursued that better balance configuration flexibility with access time and energy consumption.

## Acknowledgments

The authors would like to thank the referees for numerous valuable comments. This work was supported in part by NSF grants CDA–9401142, EIA–9972881, EIA–0080124, CCR–9702466, CCR–9701915, CCR–9811929, CCR–9988361, and CCR–9705594; by DARPA/ITO under AFRL contract F29601-00-K-0182; and by an external research grant from DEC/Compaq.

## References

[1] V. Agarwal, M.S. Hrishikesh, S. Keckler, and D. Burger, "Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures," in *Proceedings of the 27th ISCA*, 2000.

[2] D.H. Albonesi, "Dynamic IPC/clock rate optimization," *Proceedings of the 25th ISCA*, pp. 282–292, June 1998.

[3] D.H. Albonesi, "Selective cache ways: On-demand cache resource allocation," *Proceedings of MICRO-32*, pp. 248–259, November 1999.

[4] Semiconductor Industry Association, "The National Technology Roadmap for Engineers," Tech. Rep., 1999.

[5] H.B. Bakoglu and J.D. Meindl, "Optimal interconnect circuits for VLSI," *IEEE Transactions on Computers*, vol. 32, no. 5, pp. 903–909, May 1985.

[6] R. Balasubramonian, D.H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, "Dynamic memory hierarchy performance optimization," *Workshop on Solving the Memory Wall Problem*, June 2000.

[7] R. Balasubramonian, D.H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, "Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures," *Proceedings of MICRO-33*, pp. 245–257, December 2000.

[8] P. Bannon, "Alpha 21364: A scalable single-chip SMP," *Microprocessor Forum*, October 1998.

[9] W.J. Bowhill et al., "Circuit implementation of a 300-MHz 64-bit second-generation CMOS Alpha CPU," *Digital Technical Journal*, vol. 7, no. 1, pp. 100–118, Special Issue 1995.

[10] D. Burger and T. Austin, "The Simplescalar toolset, version 2.0," Tech. Rep. TR-97-1342, University of Wisconsin-Madison, June 1997.

[11] A. Buyuktosunoglu, S. Schuster, D. Brooks, P. Bose, P. Cook, and D.H. Albonesi, "A Circuit Level Implementation of an Adaptive Issue Queue for Power-Aware Microprocessors," in *Proceedings of the 11th Great Lakes Symposium on VLSI*, Mar 2001.

[12] F. Dahlgren and P. Stenstrom, "On Reconfigurable On-Chip Data Caches," in *Proceedings of MICRO-24*, 1991.

[13] W.J. Dally and J.W. Poulton, *Digital System Engineering*, Cambridge University Press, Cambridge, UK, 1998.

[14] S. Dropsho, A. Buyuktosunoglu, R. Balasubramonian, D.H. Albonesi, S. Dwarkadas, G. Semeraro, G. Magklis, and M.L. Scott, "Integrating Adaptive On-Chip Storage Structures for Reduced Dynamic Power," in *11th International Conference on Parallel Architectures and Compilation Techniques*, September 2002.

[15] K.I. Farkas and N.P. Jouppi, "Complexity/performance tradeoffs with non-blocking loads," *Proceedings of the 21st ISCA*, pp. 211–222, April 1994.

[16] B. Fisk and I. Bahar, "The Non-Critical Buffer: Using Load Latency Tolerance to Improve Data Cache Efficiency," in *IEEE International Conference on Computer Design*, October 1999.

[17] J. Fleischman, "Private communication," October 1999.

[18] L. Gwennap, "PA-8500's 1.5M cache aids performance," *Microprocessor Report*, vol. 11, no. 15, November 17, 1997.

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

[20] N.P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," *Proceedings of the 17th ISCA*, pp. 364–373, May 1990.

[21] 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*, August 1997.

[22] R. Kessler, "The Alpha 21264 microprocessor," *IEEE Micro*, vol. 19, no. 2, pp. 24–36, March/April 1999.

[23] A. Kumar, "The HP PA-8000 RISC CPU," *IEEE Computer*, vol. 17, no. 2, pp. 27–32, March 1997.

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

[25] G.W. McFarland, *CMOS Technology Scaling and Its Impact on Cache Delay*, Ph.D. thesis, Stanford University, June 1997.

[26] G.W. McFarland and M. Flynn, "Limits of scaling MOSFETS," Tech. Rep. CSL-TR-95-62, Stanford University, November 1995.

[27] T.C. Mowry, M.S. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," *Proceedings of ASPLOS-V*, pp. 62–73, October 1992.

[28] P. Ranganathan, S. Adve, and N.P. Jouppi, "Reconfigurable caches and their application to media processing," *Proceedings of the 27th ISCA*, pp. 214–224, June 2000.

[29] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren, "Supporting dynamic data structures on distributed memory machines," *Transactions on Programming Languages and Systems*, Mar. 1995.

[30] T. Romer, W. Ohlrich, A. Karlin, and B. Bershad, "Reducing TLB and Memory Overhead Using Online Superpage Promotion," in *Proceedings of the 22nd ISCA*, 1995.

[31] G.S. Sohi, "Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers," *IEEE Transactions on Computers*, vol. 39, Mar 1990.

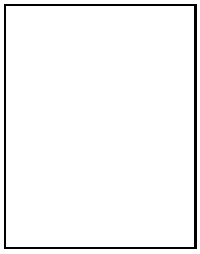
[32] S. Srinivasan, R. Ju, A. Lebeck, and C. Wilkerson, "Locality vs. Criticality," in *Proceedings of the 28th ISCA*, July 2001, pp. 132–143.

[33] S.T. Srinivasan and A.R. Lebeck, "Load Latency Tolerance in Dynamically Scheduled Processors," *Journal of Instruction-Level Parallelism*, vol. 1, Oct 1999.

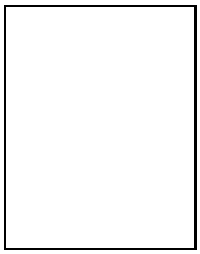
[34] A. Veidenbaum, W. Tang, R. Gupta, A. Nicolau, and X. Ji, "Adapting Cache Line Size to Application Behavior," in *Proceedings of ICS*, 1999.

[35] S. Yang, M.D. Powell, B. Falsafi, K. Roy, and T.N. Vijaykumar, "An Integrated Circuit/Architecture Approach to Reducing Leakage in Deep-Submicron High-Performance I-Caches," in *Proceedings of HPCA-7*, Jan 2001.

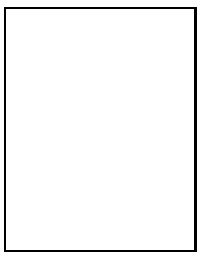
[36] K.C. Yeager, "The Mips R10000 superscalar microprocessor," *IEEE Micro*, vol. 16, no. 2, pp. 28–41, April 1996.



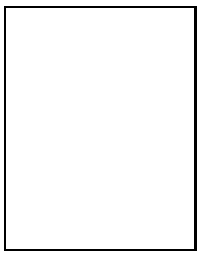
**Rajeev Balasubramonian** is a Ph.D. candidate at the Department of Computer Science at the University of Rochester. His research focuses on high performance computer architectures, given the constraints of wire-delay dominated future technologies. He received his B.Tech in Computer Science and Engineering at the Indian Institute of Technology, Bombay in 1998, and his M.S. in Computer Science from the University of Rochester in 2000.



**David H. Albonesi** is an Associate Professor of Electrical and Computer Engineering at the University of Rochester and Co-Director of the Advanced Computer Architecture Laboratory. He received his BSEE from the University of Massachusetts Amherst in 1982, his MSEE from Syracuse University in 1986, and his PhD in Electrical and Computer Engineering from the University of Massachusetts Amherst in 1996. Prior to receiving his Ph.D., he held technical and management leadership positions for 10 years at IBM Corporation (1982-86) and Prime Computer, Incorporated (1986-1992). The primary focus of his industry work was on the microarchitecture of low-latency, high-bandwidth memory hierarchies for high performance processors, the design of shared memory multiprocessor systems, and the development and application of architectural evaluation and hardware emulation tools. At Rochester, he leads the Complexity-Adaptive Processing (CAP) project and is also conducting research in the design and application of dynamic data dependence tracking hardware, multithreaded architectures, VLIW architectures for voice and video applications, adaptive clustered microarchitectures, and power-efficient highly-available systems. Dr. Albonesi received a National Science Foundation CAREER Award and IBM Faculty Partnership Awards in 2001 and 2002. He co-founded the Workshop on Complexity-Effective Design that has been held the last three years at the International Symposium on Computer Architecture. He holds five U.S. patents and is a Senior Member of the IEEE.



**Alper Buyuktosunoglu** received the B.S. degree in Electrical and Electronics Engineering from Middle East Technical University, Ankara, Turkey in 1998, with honors, and the M.S. degree in Electrical and Computer Engineering from the University of Rochester, Rochester, New York, in 1999. He is pursuing his Ph.D. degree at the University of Rochester. He is currently a research engineer at the IBM T. J. Watson Research Center. His research interests include: high-performance, low power computer architectures and digital microelectronic design. He is a member of the IEEE.



**Sandhya Dwarkadas** Sandhya Dwarkadas is an Associate Professor of Computer Science at the University of Rochester. She received the B.Tech. degree in Electrical and Electronics Engineering from the Indian Institute of Technology, Madras, India, in 1986, and her M.S. and Ph.D. in Electrical and Computer Engineering from Rice University in 1989 and 1993, respectively. From 1992-1996, she was a research scientist in the Computer Science department at Rice University. Her research interests lie in computer architecture, parallel and distributed systems, compiler/architecture/run-time interaction, simulation methodology, and performance evaluation. The projects she has worked on include the Rice Parallel Processing Testbed, an execution-driven simulation system, the design of Willow, a high-performance parallel architecture, FASTLINK, a software package for sequential and parallel genetic linkage analysis, and TreadMarks and Cashmere, software distributed shared memory systems. Her current focus is on adaptive high-performance and energy-efficient architectures as well as support for shared state in distributed systems. Dr. Dwarkadas has been the recipient of an NSF CISE Experimental Science Postdoctoral Fellowship (1993-1995), and an NSF CAREER Award for junior faculty (1997-2000). She is also associate editor of the IEEE Transactions on Parallel and Distributed Systems.