Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures[*]

Rajeev Balasubramonian[†], David Albonesi[‡], Alper Buyuktosunoglu[‡], and Sandhya Dwarkadas[†]
[†] Department of Computer Science
[‡] Department of Electrical and Computer Engineering
University of Rochester

Abstract

Conventional microarchitectures choose a single memory hierarchy design point targeted at the average application. In this paper, we propose a cache and TLB layout and design that leverages repeater insertion to provide dynamic low-cost configurability trading off size and speed on a per application phase basis. A novel configuration management algorithm dynamically detects phase changes and reacts to an application's hit and miss intolerance in order to improve memory hierarchy performance while taking energy consumption into consideration. When applied to a two-level cache and TLB hierarchy at 0.1μm technology, the result is an average 15% reduction in cycles per instruction (CPI), corresponding to an average 27% reduction in memory-CPI, across a broad class of applications compared to the best conventional two-level hierarchy of comparable size. Projecting to sub-.1 μm technology design considerations that call for a three-level conventional cache hierarchy for performance reasons, we demonstrate that a configurable L2/L3 cache hierarchy coupled with a conventional L1 results in an average 43% reduction in memory hierarchy energy in addition to improved performance.

1 Introduction

The performance of general purpose microprocessors continues to increase at a rapid pace. 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 [13]. Despite these advances, several impending bottlenecks threaten to slow the pace at which future performance improvements can be realized. Arguably the single biggest potential bottleneck for many applications in the future will be high memory latency and the lack of sufficient memory bandwidth. Although advances such as non-blocking caches [10] and hardware and software-based prefetching [14, 21] can reduce latency in some cases, the underlying structure of the memory hierarchy upon which these approaches are implemented may ultimately limit their effectiveness. 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. Although several well-known organizational techniques can be used to reduce the power dissipation in on-chip memory structures, 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 [6]) requires that these structures be effectively used so as not to needlessly waste chip power. Thus, new approaches that improve performance in a more energy-efficient manner than conventional memory hierarchies are needed to prevent the memory system from fundamentally limiting future performance gains or exceeding power constraints.

The most commonly implemented memory system organization is likely the familiar 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 [24]), 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 tradeoff between optimizing hit time and miss time. Although this approach works well for many common desktop applications and benchmarks, 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 perfor-

[*] This work was supported in part by NSF grants CDA–9401142, EIA-9972881, 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.

0-7695-0924-X/2000/$10.00 © 2000 IEEE

mance 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 [12, 17, 18]) may perform better and be more energy-efficient than a two-level hierarchy for the same total amount of memory. For similar reasons, the PA-8X00 series also implements a large, single-level TLB. Because the TLB and cache are accessed in parallel, a larger TLB can be implemented without impacting hit time in this case due to the large L1 caches that are implemented.

The fundamental issue in current approaches is that no one memory hierarchy organization is best suited for each application. Across a diverse application mix, 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. In this paper, we present a configurable cache and TLB orchestrated by a configuration algorithm that can be used 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 manner. Our approach monitors cache and TLB usage by detecting phase changes using miss rates and branch frequencies, and improves performance by properly balancing hit latency intolerance with miss latency intolerance dynamically during application execution (using CPI as the ultimate performance metric). 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. Finally, energy-aware modifications to the configuration algorithm are implemented 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 detail, 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.

Recently, Ranganathan, Adve, and Jouppi [22] 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.

This paper significantly expands upon our results in [5] that addressed only performance in a limited manner for one technology point (0.1μm) using a different (more hardware intensive) configuration algorithm. In this paper, we explore the application of the configurable hierarchy as a L1/L2 replacement in 0.1μm technology, and as an L2/L3 replacement for a 0.035μm feature size. For the former, we demonstrate an average 27% improvement in memory performance, which results in an average 15% improvement in overall performance as compared to a conventional memory hierarchy. Furthermore, the energy-aware enhancements that we introduce bring memory energy dissipation in line with a conventional organization, while still improving memory performance by 13% relative to the conventional approach. For 0.035μm geometries, where the prohibitively high latencies of large on-chip caches [1] call for a three-level conventional hierarchy for performance reasons, we demonstrate that a configurable L2/L3 cache hierarchy coupled with a conventional L1 reduces overall memory energy by 43% while even slightly increasing performance. This latter result demonstrates that because our configurable approach significantly improves memory hierarchy efficiency, it can serve as a partial solution to the significant power dissipation challenges facing future processor architects.

The rest of this paper is organized as follows. The cache and TLB architectures are described in Section 2 including the modifications necessary to enable dynamic reconfiguration. In Section 3, we discuss the dynamic selection mechanisms, including the counter hardware required and the configuration management algorithms. In Sections 4 and 5, we describe our simulation methodology and present a performance and energy dissipation comparison with conventional multi-level cache and TLB hierarchies for the two technology design points. Finally, we conclude in Section 6.

2 Cache and TLB Circuit Structures

In this section, we describe the circuit structures of the conventional and configurable caches and TLBs that we consider. We also describe two different approaches for using configurable caches as replacements for conventional on-chip cache hierarchies.

2.1 Configurable Cache Organization

The cache and TLB structures (both conventional and configurable) that we model follow that described by Mc-

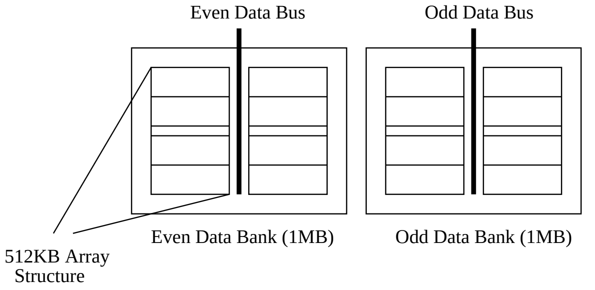

Figure 1. The overall organization of the cache data arrays

Farland in his thesis [19]. 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 start with a conventional 2MB data cache that is organized both for fast access time and energy efficiency. As is shown in Figure 1, the cache is structured as two 1MB interleaved banks[1] in order to provide sufficient memory bandwidth for the four-way issue dynamic superscalar processor that we simulate. In order to reduce access time and energy consumption, each 1MB bank is further divided into two 512KB SRAM 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 2 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. 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 [11].

In order to provide configurability while retaining fast access times, we implement several modifications to McFarland's baseline design as shown in Figure 2:

- 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 2 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, we estimate the additional area from adding repeater switches to electrically isolate wordlines to be 7%. In addition, due to the large capacity (and resulting long wordlines) of each cache structure, a faster propagation delay is achieved with these buffered wordlines compared with unbuffered lines. Moreover, because local wordline drivers are required in a conventional cache, the extra drivers required to isolate ways within a subarray 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% in comparison with a cache with no repeaters for the simulated benchmarks.

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

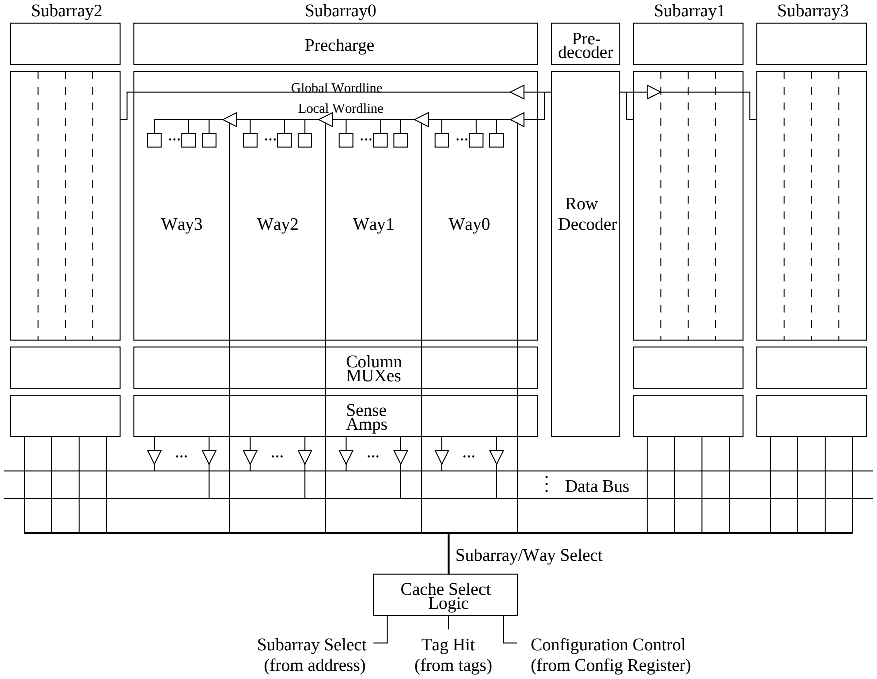

Figure 2. The organization of the data array section of one of the 512KB cache structures

2.2 Configurable Cache Operation

With these modifications, the cache behaves as a *virtual two-level, physical one-level* non-inclusive cache hierarchy, with the sizes, associativities, and latencies of the two levels dynamically chosen. In other words, we have designed a single large cache organization to serve 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 the two sections is changed on half-cycle increments according to the timing of each configuration (and assuming a 1 GHz processor). 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 [16], 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.

When used as a replacement for a conventional L1/L2 on-chip cache hierarchy, the possible configurations are shown in Figure 3. 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 3). 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 compara-

Subarray/Way Allocation (L1 or L2)

Cache Configuration				Subarray 2				Subarray 0				Subarray 1				Subarray 3			
	L1 Size	L1 Assoc	L1 Acc Time	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***

Figure 3. 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 tradeoffs allowable.

ble performance yet potentially save considerable energy. These configurations are used in an energy-aware mode of operation as described in Section 3.

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*). In cases where subarrays two and three are disabled, the high-order *Subarray Select* signal is used as a tag bit. This extra tag bit is stored on all accesses in order to detect mis-maps. 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 sub-0.1μm technologies, the long access latencies of a large on-chip L2 cache [1] may be prohibitive for those applications which 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 5 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 0.1μm.

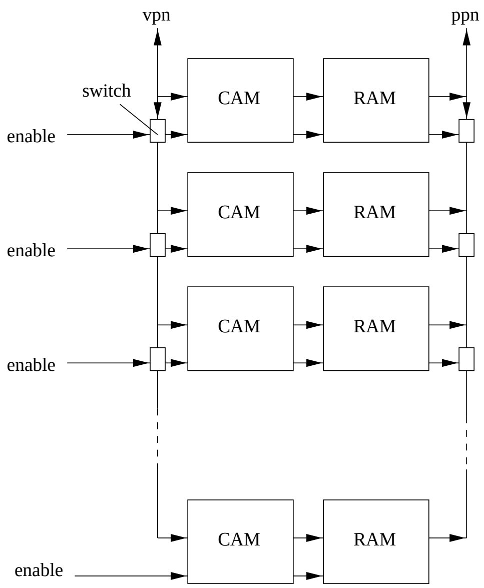

Figure 4. The organization of the configurable TLB

2.3 Configurable TLB Organization

Our 512-entry, fully-associative TLB can be similarly configured as shown in Figure 4. 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, TLB misses result in a second access but to the backup portion of the TLB.

3 Dynamic Selection Mechanisms

In this section, we first describe selection mechanisms for the configurable cache and TLB when used as a replacement for a conventional L1/L2 on-chip hierarchy. In the last subsection, we discuss the mechanisms as applied to a configurable L2/L3 cache hierarchy coupled with a conventional fixed-organization L1 cache.

Our configurable cache and TLB approach makes it possible to pick 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.

3.1 Search Heuristics

Large L1 caches have a high hit rate, but also have higher access times. To arrive at the cache configuration that is the optimal trade-off point between the cache hit and miss times, we use a simple mechanism that uses past history to pick a size for the future, based on CPI as the performance metric.

Our initial scheme is tuned to improve performance and thus explores the following five cache configurations: direct-mapped 256KB 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 in our simulations), we examine a set of hardware counters. These hardware counters tell us the miss rate, the IPC, 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 and pick an appropriate size.

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 case) 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 in the previous interval. When there is a change, we switch to the unstable state, return to the smallest L1 cache configuration and start exploring again. The pseudocode for the mechanism is listed below.

```
if (state == STABLE)
    if ((num_miss-last_num_miss) < m_noise
        && (num_br-last_num_br) < br_noise)
        decr m_noise, br_noise;
    else
        cache_size = SMALLEST;
        state = UNSTABLE;

if (state == UNSTABLE)
    record CPI;
    if ((miss_rate > THRESHOLD)
        && (cache_size != MAX))
        cache_size++;
    else
        cache_size = that with best CPI;
        state = STABLE;
        if (cache_size == prev_cache_size)
            incr br_noise, m_noise;
```

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 change this dynamically. If an exploration phase results in picking the same cache size as before, the noise threshold is increased to discourage such needless explorations. Likewise, every interval spent in the stable state causes a slight decrement in the noise threshold in case it had been set to too high a value.

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. In addition, we keep track of how many intervals are spent in stable and unstable states. If it turns out that we are spending too much time exploring, we conclude that the program behavior is not suited to an interval-based scheme and simply remain fixed at the smallest sized cache.

Our earlier experiments [5] used a novel hardware design to estimate the hit and miss latency intolerance of an application's phase (which our selection mechanism is attempting to minimize). These estimates were then used to detect phase changes as well as to guide exploration. As our results show in comparison to those in [5], the additional complexity of the hardware is not essential to obtaining good performance. Presently, we envision that the selection mechanism would be implemented in software. Every 100K cycles, a low-overhead software handler will be 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 mechanism. We estimated the code size of the handler to be only 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 overhead, we need roughly 9 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.

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 cleared at start of an interval). 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 larger one million cycle interval so that an accurate estimate of TLB usage could be obtained. A smaller interval size could result in a spuriously high TLB miss rate over some intervals, and/or low TLB usage.

3.2 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. 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 most important routines, we only monitor those subroutines whose invocations exceed a certain threshold of 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 can be determined 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 played a role in determining the configuration of the caller. In the *nested* scheme, each subroutine collects statistics only for the period when it is the top of the subroutine call stack. Thus, every single subroutine invocation is looked upon as a possible change in phase.

Because the simpler non-nested approach generally outperformed the nested scheme, we only report results for the former in Section 5.

3.3 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 potential approach is to serially access the tag and data arrays of the L1 data cache. Conventional L1 caches always perform parallel tag and data lookup 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 lookup 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.

3.4 L2/L3 Reconfiguration

The selection mechanism for the L2/L3 reconfiguration is very 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 lookup 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. Much of 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 as well (within 95% in our simulations) as the best performing cache during the exploration, thus reducing the number of transfers between the L2 and L3.

4 Evaluation Methodology

4.1 Simulation Methodology

We used Simplescalar-3.0 [8] for the Alpha AXP instruction set to simulate an aggressive 4-way superscalar out-of-order processor. The architectural parameters used in the simulation are summarized in Table 1.

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.

For both configurable and conventional TLB hierarchies, a TLB miss at the first level results in a lookup 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.

Fetch queue entries	8
Branch predictor	comb. of bimodal & 2-level gshare; bimodal/Gshare Level1/2 entries - 2048, 1024 (hist. 10), 4096 (global), resp.; Combining pred. entries - 1024; RAS entries - 32; BTB - 2048 sets, 2-way
Branch mispred. latency	8 cycles
Fetch, decode, issue width	4
RUU and LSQ entries	64 and 32
L1 I-cache	2-way; 64KB (0.1μm), 32KB (0.035μm)
Memory latency	80 cycles (0.1μm), 114 cycles (0.035μm)
Integer ALUs/mult-div	4/2
FP ALUs/mult-div	2/1

Table 1. Architectural parameters

4.2 Benchmarks

We have used a variety of benchmarks from SPEC95, SPEC2000, and the Olden suite [23]. These particular 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. The window size was chosen to be large enough to accommodate at least one outermost iteration of the program, where applicable. A further million instructions were simulated in detail to prime all structures before starting the performance measurements. Table 2 summarizes the benchmarks and their memory reference properties (the L1 miss rate and load frequency).

4.3 Timing and Energy Estimation

We investigated two future technology feature sizes: 0.1 and 0.035μm. For the 0.035μm 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 0.1μm design point, we use the cache and TLB timing model developed by McFarland [19] to estimate timings for both the configurable cache and TLB, and the caches and TLBs of a conventional L1/L2 hierarchy. 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

Benchmark	Suite	Datasets	Simulation window (instrs)	64K-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%

Table 2. Benchmarks

scaling down to 0.1μm technology [20]. 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 [9]. Repeaters are used in the configurable cache as well as in the conventional L1 cache whenever they reduce wire propagation delay. The energy dissipation of these repeaters was accounted for as well, and they add only 2-3% to the total cache energy.

We estimate cache and TLB energy dissipation using a modified version of the analytical model of Kamble and Ghose [15]. 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 0.1μm 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 on-chip L2 cache [7]. In addition, 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 both fast access time and low energy dissipation.

Detailed event counts were captured during SimpleScalar simulations of each benchmark. These event counts include all of the operations that occur for the configurable cache as well as all TLB events, and are used to obtain final energy estimations.

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

Table 3. Simulated L1/L2 configurations

4.4 Simulated Configurations

Table 3 shows the conventional and dynamic L1/L2 schemes that were simulated. We compare our dynamic schemes with three conventional configurations which 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 which 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 lookup time.

For L2/L3 reconfiguration, we compare our interval-

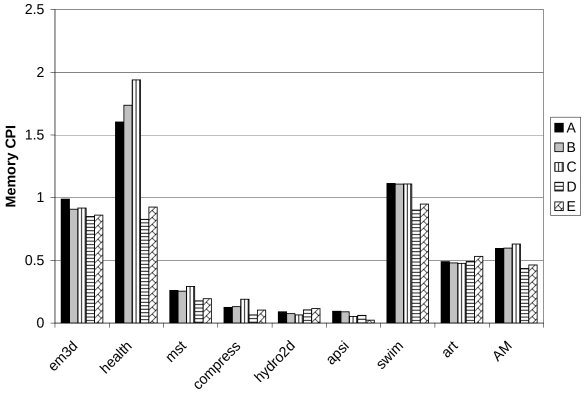

Figure 5. Memory CPI for conventional (A, B, and C), interval-based (D), and subroutine-based (E) configurable schemes

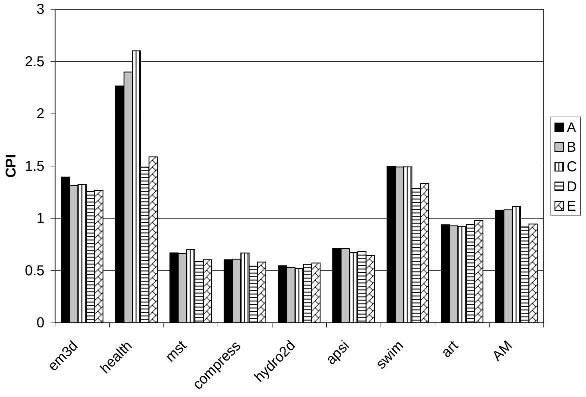

Figure 6. CPI for conventional (A, B, and C), interval-based (D), and subroutine-based (E) configurable schemes

	Cache contribution	TLB contribution	Cache explorations	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

Table 4. Contribution of the cache and the TLB to speedup or slowdown in the dynamic scheme and the number of explorations

based configurable cache with a conventional three-level on-chip hierarchy. In both, the L1 cache is 32KB two-way set associative with a three cycle latency, reflecting the smaller L1 caches and increased latency likely required at 0.035μm 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.

5 Results

We first evaluate the performance and energy dissipation of the L1/L2 configurable schemes versus the three conventional approaches using delay and energy values for 0.1μm geometries. We then demonstrate how L2/L3 reconfiguration can be used at finer 0.035μm geometries to dramatically improve energy efficiency relative to a conventional three-level hierarchy but with no compromise of performance.

5.1 L1/L2 Performance Results

Figures 5 and 6 show the memory CPI and total CPI, respectively, 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 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 CPI) by 27%, with roughly equal cache and TLB contributions as is shown in Table 4. 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, 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's 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 4). 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.

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. Because this happens early enough in case of art (the simulation window is also much larger), art shows no overall performance degradation, while hydro2d has a slight 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 comparing the interval and subroutine-based schemes, we conclude that the simpler interval-based scheme usually outperforms the subroutine-based approach. The most notable exception is apsi, which has inconsistent behavior across intervals (as indicated by the large number of explorations in Table 4), causing it to thrash between a 256KB L1 and a 768KB L1. The subroutine-based scheme significantly improves performance relative to the interval-based approach as each subroutine invocation within apsi exhibits consistent behavior from invocation to invocation. Yet, due to the overall results and the additional complexity of the subroutine-based scheme, the interval-based scheme appears to be the most practical choice and is the only scheme considered in the rest of our analysis.

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.

These 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 tradeoffs, we varied the processor pipeline speed relative to the memory latencies (keeping the memory hierarchy latency fixed). The results in terms of performance improvement were similar for 1 (our base case), 1.5, and 2 GHz processors.

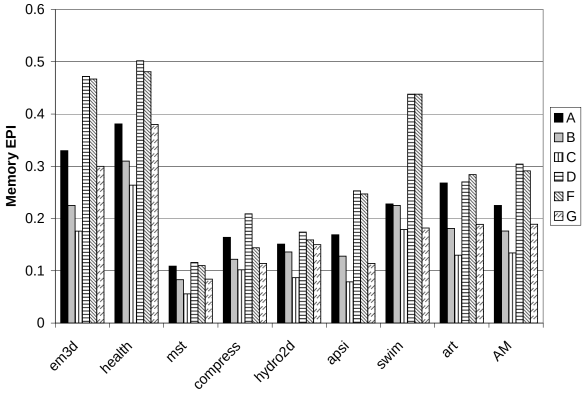

Figure 7. Memory EPI (in nanoJoules) for conventional (A, B, and C), interval-based (D), and energy-aware (F and G) configurable schemes

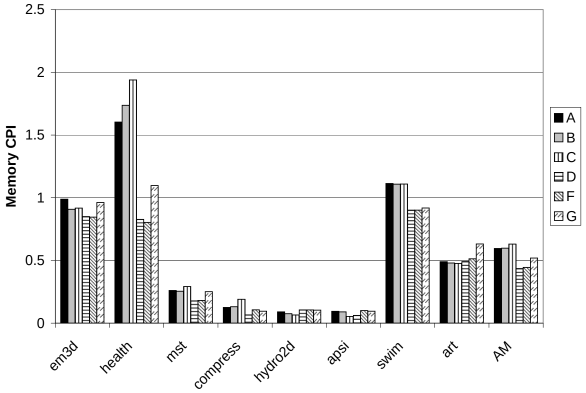

Figure 8. Memory CPI for conventional (A, B, and C), interval-based (D), and energy-aware (F and G) configurable schemes

5.2 Energy-Aware Configuration Results

We focus here on the energy consumption of the on-chip memory hierarchy (including that to drive the off-chip bus). The memory energy per instruction (memory EPI, with each energy unit measured in nanoJoules) results of Figure 7 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 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 8), 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). Thus, the energy-aware approach may be used to provide more modest performance improvements 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.

5.3 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-0.1μm 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 0.035μm technology to illustrate how dynamic L2/L3 cache configuration can match the performance of a conventional three-level hierarchy while dramatically reducing energy dissipation.

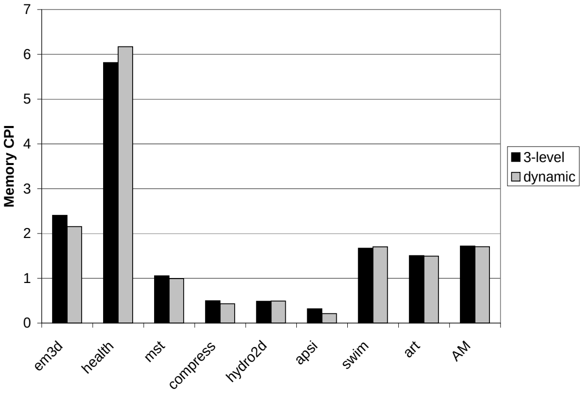

Figure 9. Memory CPI for conventional three-level and dynamic cache hierarchies

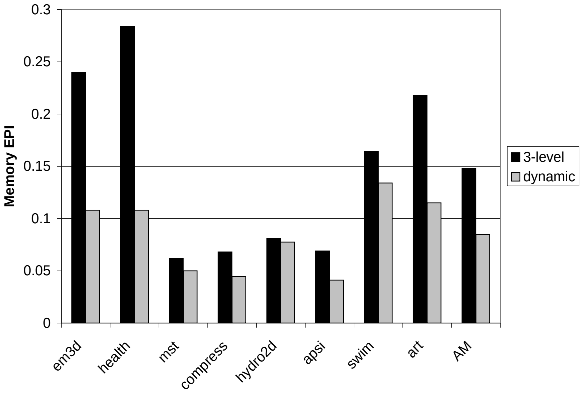

Figure 10. Memory EPI (in nanoJoules) for conventional three-level and dynamic cache hierarchies

Figures 9 and 10 compare the performance and energy, respectively, 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 43% 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.

6 Conclusions

We have described a novel configurable cache and TLB as an alternative to conventional cache hierarchies. Repeater insertion is leveraged to enable dynamic cache and TLB configuration, with an organization that allows for dynamic speed/size tradeoffs while limiting the impact of speed changes to within the memory hierarchy. Our configuration management algorithm is able to dynamically examine the tradeoff between an application's hit and miss intolerance using CPI as the ultimate metric to determine appropriate cache size and speed. At 0.1μm 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. Projecting to 0.035μm technologies and a 3-level cache hierarchy, we show improved performance with an average 43% reduction in memory hierarchy energy when compared to a conventional design. This latter result demonstrates that because our configurable approach significantly improves memory hierarchy efficiency, it can serve as a partial solution to the significant power dissipation challenges facing future processor architects.

Future work includes investigating the use of compiler support for applications where an interval-based scheme is unable to capture the phase changes (differing working sets) 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. Finally, improvements at the circuit and microarchitectural levels will be pursued that better balance configuration flexibility with access time and energy consumption.

References

[1] V. Agarwal, M. Hrishikesh, S. Keckler, and D. Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. *Proceedings of the 27th International Symposium on Computer Architecture*, pages 248–259, June 2000.

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

[3] D. Albonesi. Selective cache ways: On-demand cache resource allocation. *Proceedings of the 32nd International Symposium on Microarchitecture*, pages 248–259, November 1999.

[4] S. I. Association. The National Technology Roadmap for Engineers. Technical report, 1999.

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

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

[7] 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.

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

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

[10] K. Farkas and N. Jouppi. Complexity/performance tradeoffs with non-blocking loads. *Proceedings of the 21st International Symposium on Computer Architecture*, pages 211–222, April 1994.

[11] J. Fleischman. Private communication. October 1999.

[12] L. Gwennap. PA-8500's 1.5M cache aids performance. *Microprocessor Report*, 11(15), November 17, 1997.

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

[14] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *Proceedings of the 17th International Symposium on Computer Architecture*, pages 364–373, May 1990.

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

[16] R. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.

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

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

[19] G. McFarland. *CMOS Technology Scaling and Its Impact on Cache Delay*. PhD thesis, Stanford University, June 1997.

[20] G. McFarland and M. Flynn. Limits of scaling MOSFETS. Technical Report CSL-TR-95-62, Stanford University, November 1995.

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

[22] P. Ranganathan, S. Adve, and N. Jouppi. Reconfigurable caches and their application to media processing. *Proceedings of the 27th International Symposium on Computer Architecture*, pages 214–224, June 2000.

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

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