Dynamic Memory Hierarchy Performance Optimization*

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

Abstract

Although microprocessor performance continues to increase at a rapid pace, the growing processor-memory speed gap threatens to limit future performance gains. In this paper, we propose a novel configurable cache and TLB as an alternative to conventional two-level hierarchies. This organization leverages repeater insertion to provide low-cost configurability of size and speed. A novel configuration management algorithm dynamically measures hit and miss intolerance over intervals of instruction execution in order to tailor the cache and TLB organizations on-the-fly to improve memory hierarchy performance. The result is an average 14% improvement in IPC and a speedup of up to 1.55 across a broad class of applications compared to a conventional two-level hierarchy of identical total size.

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 [9]. 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 [5] and hardware and software-based prefetching [10, 16] can reduce latency in some cases, the underlying structure of the memory hierarchy upon which these approaches are implemented may ultimately limit their effectiveness. Thus, new approaches that provide for lower latency and higher bandwidth than conventional memory hierarchies are needed to prevent the memory system from fundamentally limiting future performance gains.

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 [20]), 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 set exceeds the L1 capacity may spend 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 [8, 12, 13]) may perform better 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.

One 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 due to a mismatch between the memory system requirements of the application and the memory hierarchy implementation.

Previous approaches to this problem [1, 2] have exploited the partitioning of hardware resources to enable/disable parts of the cache under software control, but in a limited manner. In [1], a preliminary analysis of a cache hierarchy using the approach was presented. The design assumed a two-level on-chip cache in which a set of partitions could be allocated between L1 and L2 as needed. However, the issues of how to practically implement such a design were not addressed in detail,

*This work was supported in part by NSF grants CDA–9401142, EIA-9972881, CCR–9702466, CCR–9701915, CCR–9811929, and CCR–9705594; and an external research grant from DEC/Compaq.

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 clock frequency with increases in cache size limited the performance improvement that could be realized.

In [2], it was demonstrated that cache energy dissipation could be reduced by enabling all of the cache ways when required to achieve high performance, but enabling only a subset of the ways when cache demands were more modest. As in [1], a perfect selection algorithm was assumed and a single overall-best configuration was used for each application. In addition, the cache was partitioned into a separate subarray for each cache way, whereas in practice, a different subarray organization may be used that allows for conditional clocking of subarrays.

In this paper, we present what we contend to be a superior alternative to static two-level cache and TLB hierarchies: a configurable cache and TLB orchestrated by a configuration algorithm that seeks to maximize performance. Unlike the approach in [2] and similar to the approach in [1], parts of the cache that are not used as an L1 cache serve as a backup L2 cache. In effect, what we have is 2MB of on-chip cache real-estate that is dynamically configured between L1 and L2, with the goal of arriving at the optimal tradeoff between L1 hit time and L1 miss time. Instead of changing the clock rate as proposed in [1], 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. Furthermore, we propose a novel hardware design that monitors cache and TLB usage and application latency tolerance at regular intervals, and improves performance by properly balancing hit intolerance with miss intolerance dynamically during application execution.

The rest of this paper is organized as follows. Section 2 describes the layout of the architecture, and how it enables dynamic reconfiguration. Section 3 describes the dynamic selection mechanisms, including the hardware counters required and the hardware-based configuration management algorithm. Sections 4 and 5 describe our simulation methodology and present a performance comparison with conventional two-level cache and TLB hierarchies. We conclude in Section 6.

2 Cache and TLB Architecture

Our cache and TLB structures follow that described by McFarland in his thesis [14]. 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 use a 2MB data cache that is two-way banked and interleaved on a word basis in order to provide enough memory bandwidth for the four-way dynamic superscalar processor that we simulate. This allows us to access two adjacent words in a cache block at the same time. In order to reduce access time and energy consumption, each bank is implemented as two 512KB SRAM structures, one of which is selected on each bank access. The data array section of the structure is shown in Figure 1 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 order to reduce energy dissipation, two address bits (*Subarray Select*) are used to select only one of the four subarrays on each access. 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 [7].

In order to provide adaptivity while retaining fast access times, we implement several modifications to McFarland's baseline design as shown in Figure 1:

- 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 contrast, 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.
- Repeaters 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.
- 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.*. We organize the bits according to ways as shown in Figure 1 in order to increase the number of configuration options.
- Repeaters are used in the local wordlines to electrically isolate each way in a subarray. The result is that

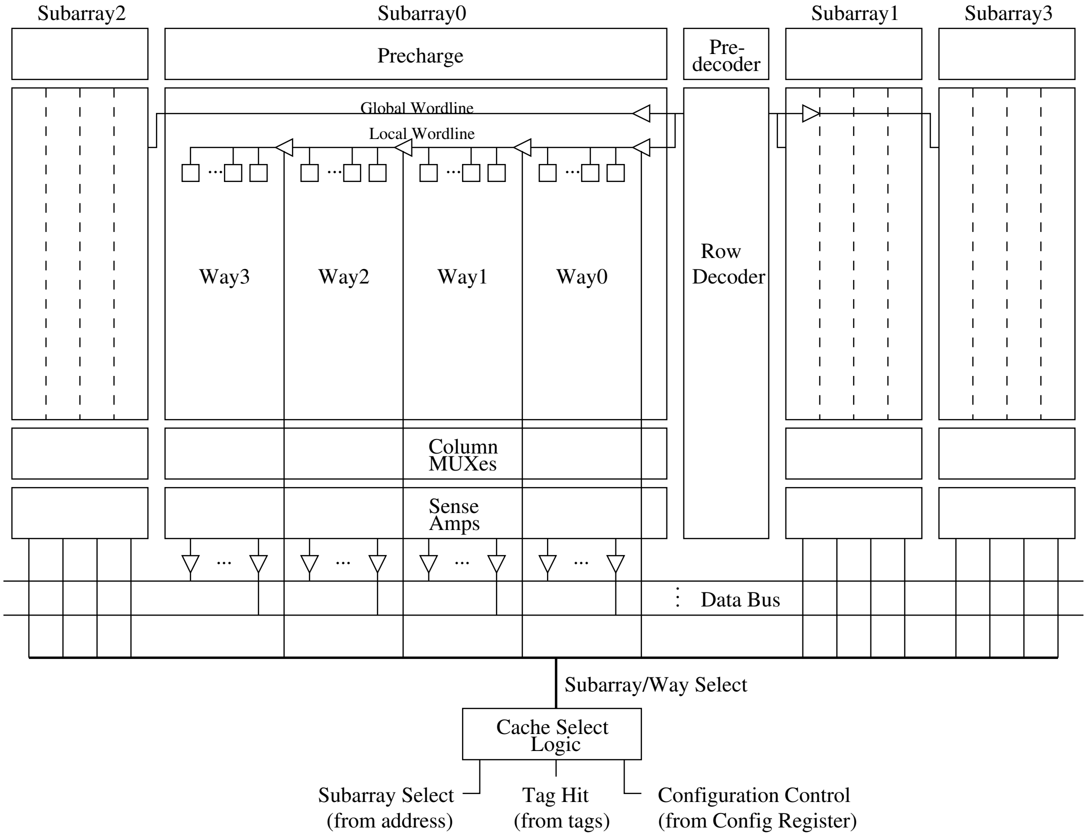

Figure 1: The organization of the data array section of one of the 512KB cache structures

the presence of additional ways does not impact the delay of the fastest 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.

We estimated the additional area from adding minimum size repeaters to electrically isolate wordlines to be just over 6%. In addition, due to the large capacity of each cache structure, each local wordline is roughly 2.75mm in length (due to the size of the cache) at 0.1 μm technology (assumed throughout this paper), and therefore 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 repeaters increases the total memory hierarchy energy dissipation by 2-3% in comparison with a cache with no repeaters for the simulated benchmarks.

Figure 2 shows the cache configurations possible in our design. Although multiple subarrays are enabled in an organization, only one is selected each cycle according to the *Subarray Select* field of the address. However, the number of enabled ways within each subarray that are initially enabled on an access may be varied. When a miss in the enabled ways is detected, all tag subarrays and ways are read. This permits hit detection to data in disabled ways within enabled subarrays as well as to data that has been mismapped due to reconfiguration (discussed below). When this occurs, the data in the enabled way (which has already been read out and placed into a buffer) is swapped with the data in the disabled way. In the case of a miss to both enabled and disabled ways, the displaced block in the enabled way is placed into one of the disabled ways or subarrays. This prevents thrashing in the case of low-associative organizations. In effect, the cache behaves like a two-level exclusive cache, with the sizes of L1 and L2 being dynamically chosen. 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

Enabled Subarrays/Ways

Cache Configuration				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								E	E							
512-2	512KB	2 way	2.5							E	E	E	E						
768-3	768KB	3 way	2.5						E	E	E	E	E	E					
1024-4	1024KB	4 way	3.0					E	E	E	E	E	E	E	E				
512-1	512KB	1 way	3.0				E				E	E				E			
1024-2	1024KB	2 way	3.5			E	E			E	E	E	E			E	E		
1536-3	1536KB	3 way	4.0		E	E	E		E	E	E	E	E	E		E	E	E	
2048-4	2048KB	4 way	4.5	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E

Figure 2: Possible cache organizations that can be configured shown by the ways that are initially enabled (as L1) denoted with an E under a particular column. Only one of the four SRAM structures is shown; the ways enabled within each structure are identical. Abbreviations for each organization are listed to the left of the size and associativity of the L1 section, while 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 performance/energy tradeoffs allowable.

another at another time (resulting in the mismap alluded to earlier). 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 mismaps. Mismapped data is handled the same way as a level one miss and level two hit, *i.e.*, it results in a swap. Our simulations indicate that such events are infrequent.

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 save considerable energy. We focus in this paper on performance, and hence currently do not use these two configurations. Hence, if the application cannot tolerate level one cache misses, the size is increased progressively from 256KB 1-way to 768KB 3-way to 1MB 4-way and then finally onto 1.5MB 3-way and 2MB 4-way. From a performance perspective, the 512KB 2-way configuration provides no advantage over the 768KB 3-way configuration (due to their identical access times in cycles) and thus this 512KB configuration is not used.

Our 512-entry, fully-associative TLB can be similarly configured as shown in Figure 3. 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. Repeaters 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. Unlike the cache design, data is not swapped between the primary and backup portions.

The performance improvement gained in [1] was limited by the fact that the clock rate of the chip was increased whenever the cache size was increased beyond its minimum size. Thus, all pipeline stages were forced to operate at this slowed down rate, which mitigated the performance benefits of increasing cache size. To remedy this situation, we vary the latency of the cache and TLB access on half-cycle increments according to the timing of each configuration, assuming a two cycle access for the minimum size direct-mapped 256kB configuration (Refer to Figure 2 for timing values.). Half cycle increments are necessary 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 [11], 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,

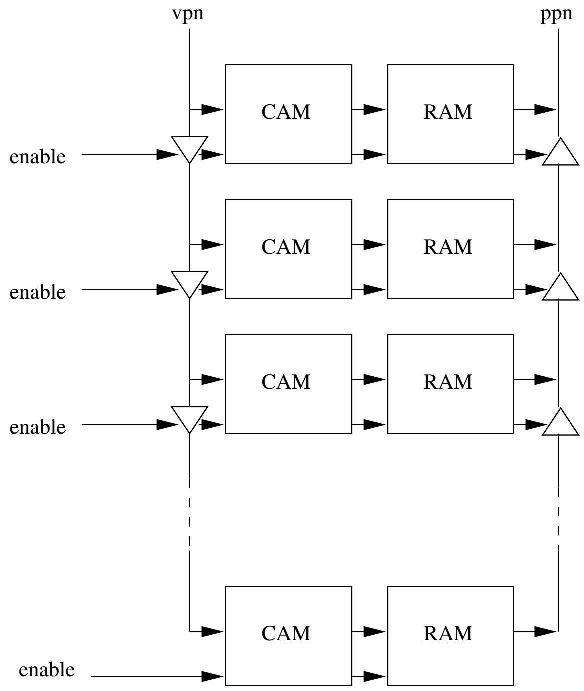

Figure 3: The organization of the configurable TLB

and no clock synchronization is necessary. However, because we assume that the processor uses only one phase of the clock, cache data that is latched using the alternate phase is used by the processor a half-cycle later. In addition, the control logic that determines when an instruction that uses the result of a load should be speculatively issued (assuming a cache hit) must take into account the configuration that is enabled. Finally, we maintain a two stage cache pipeline access for each configuration. This implies a lower throughput for the larger configurations as well as for the large conventional L2 cache due to their higher latencies.

3 Dynamic Selection Mechanisms

Our cache and TLB organization makes it possible to pick an L1 cache size depending on the application's requirements. The appropriate size at any time is that which achieves the best trade-off between the access time for each cache hit and the time spent servicing cache misses. Cache miss rates give 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. In this section, we first present a metric that quantifies this effect and describe how it can be used to dynamically pick an appropriate cache size.

3.1 Latency Tolerance Metrics

The actual number of memory stall cycles incurred 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 [19], and [6] 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 caused by cache hits and misses.

We assume that the issue logic is built around the Register Update Unit (RUU) [18]. The RUU holds all instructions that are at different stages in the pipeline (queued, issued, and completed). To every entry in the RUU, in addition to the ready bits for the operands, we add two bits per operand: one specifying if the operand is produced by a load and another specifying if the load was a hit or a miss. At instruction decode time, this information can be deduced from the register map table. Every cache miss results in a broadcast of the destination register tag to the RUU, so that the entries can update the hit/miss status of their operands. Every cycle, we use this information to determine how many instructions were stalled by an outstanding load. For every instruction in the RUU that directly depends on a load, we increment an intolerance counter if (i) all operands except the operand produced by a load are ready, (ii) a functional unit is available, and (iii) there are free issue slots in that cycle. Depending on whether the load is marked as a hit or a miss, the intolerance is classified as hit or miss intolerance. 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.

The metric just described has limitations in the presence of multiple stalled instructions due to loads. Free issue slots may be mis-categorized as hit or miss intolerance if the resulting dependence chains were to converge. This mis-categorization of lack of ILP manifests itself when the converging dependence chains are of different lengths. Stalling the shorter chain for a period of time should not affect the execution time. Hence, the number of program stall cycles should be dependent on the stall cycles for the longer dependence chain. The chain in the critical path is difficult to compute at runtime. The miss and hit intolerance metrics effectively add the stalls for both chains, and in practice, seem to work well.

Since we are also interested in TLB reconfiguration, we need a metric for picking an appropriate TLB size. In our model, the pipeline stalls while a TLB miss is being serviced. Hence, the TLB miss rate serves as a reasonable approximation to the effect of the TLB on execution time. We also have a TLB usage metric that counts the number of TLB entries that are accessed since the last count.

3.2 Improving Performance Using Tolerance Information

The hit and miss intolerance counters indicate the effect of a given cache organization on actual execution time. Large caches tend to have higher hit intolerance because of the greater access time, but lower miss intolerance due to the smaller miss rate. These intolerance counters serve as a hint to indicate which cache sizes to explore and the optimal cache configuration is the one that usually has the smallest sum of hit and miss intolerance. To arrive at this configuration dynamically at runtime, we use a simple mechanism that uses past history to pick a size for the future.

We examine hit and miss intolerance values in every million cycle interval. Based on this, we pick 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 cache size, 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 cache size is chosen to be the smallest. If the miss intolerance for an interval exceeds the hit intolerance, we move to a bigger cache size in the hope that we can contain the working set and bring about a drastic drop in miss intolerance. Likewise, if the hit intolerance is greater, we move to a smaller size. For every different cache size, we keep the CPI for that interval in a table. This exploration continues until a cache size is revisited or the maximum or minimum size is reached. At this point, the table is examined to pick the cache configuration that worked best, the table is cleared, and we switch to the stable state. We continue to remain in the stable state while hit and miss intolerance 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 cache size and start exploring again. The pseudocode for the mechanism is listed below.

```
if stable
  NOISE = 0.3*(last_miss_intol+
               last_hit_intol);
  if miss_intol-last_miss_intol < NOISE
  and hit_intol-last_hit_intol < NOISE
    remain at stable;
  else
    make unstable;
    choose smallest cache;

if unstable
  update table entries;
  FACTOR = latency_of_next_size/
         latency_of_current_size - 1;
  if miss_intol > FACTOR*hit_intol
    increase size;
  else if hit_intol > miss_intol
    decrease size;
  if revisiting or
     unable to incr/decr any further
       inspect table;
       move to best size;
       make state stable;
```

The rationale behind the use of FACTOR is that the miss intolerance tends to decline sharply when the working set fits in the cache. Hence, we want to ensure that larger cache sizes are explored as long as there is the possibility that the reduction in miss intolerance is not offset by the corresponding increase in hit intolerance. Similarly, NOISE is used to incorporate some hysteresis when reacting to changes in the application behavior so as to avoid unnecessary changes in cache organization. We chose an interval of a million cycles as an appropriate checkpoint.

Clearly, this 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 cache size 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 more like that of a conventional cache. Other heuristics (such as exploring from the current size instead of moving to the smallest size) were also tried and the above heuristic worked best for most applications.

In addition to cache reconfiguration, we also change the size of the TLB. TLB size is increased if the miss rate is high enough that the TLB overhead exceeds 3% of execution time. The size is decreased if the TLB usage is less than half. At the time of changing cache and TLB size, once the appropriate sizes are chosen, the dominating (larger) latency is used to determine whether a larger cache (or TLB) would work just as well without impacting overall memory latency, and that size is used instead.

Fetch queue size	8
Branch predictor	comb. of bimodal and 2-level gshare
Bimodal size	2048
Level1	1024entries,history 10
Level2	4096 entries (global)
Combining predictor size	1024
RAS size	32
BTB	2048 sets, 2-way
Branch misprediction latency	5 cycles
Fetch, decode and issue width	4
RUU and LSQ size	64 and 32
L1 I-cache	64KB 2-way
Memory latency	66 cycles
Integer ALUs	4
Integer mult/div	2
FP ALUs	2
FP mult/div	1

Table 1: Simplescalar simulator parameters

4 Evaluation Methodology

4.1 Simulation Methodology

We used Simplescalar-3.0 [3] for the Alpha AXP instruction set to simulate an aggressive 4-way superscalar out-of-order processor. The simulation parameters are summarized in Table 1.

The data memory hierarchy is modeled in great detail. For the reconfigurable cache, the 2MB of on-chip cache is partitioned as a two-level exclusive cache, where the size of the L1 is dynamically picked. It is organized as two word-interleaved banks, each of which can service up to one cache request every cycle. It is assumed that the access is pipelined, so a fresh request can issue after half the time it takes to complete one access. The bus between the two levels and to memory, writeback buffers, and contention for the caches have also been modeled. We assume that the access to the second level is pipelined and a fresh request can issue every 4 cycles. The second level access time is 15 cycles. We picked a line size of 128 bytes because it yielded a much lower miss rate for our benchmark set than smaller line sizes. It also enabled a more optimal layout, and hence lower access times. The downside of this choice is the need for a wider bus between L1 and L2, but this investment seems worthwhile, given the much better performance that it affords (due to the lower miss rate and faster access time). As was shown in Figure 2, the minimum cache is 256KB and direct mapped, while the largest is 2MB 4-way, the access times being 2 and 4.5 cycles, respectively. Our minimum sized TLB has 64 entries, while the largest is 512. A TLB miss at the first level results in a 6 cycle lookup in the second level (no swap is done). 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. Note that the TLB is not like an inclusive 2-level TLB – the second level is never written to. It is looked up in the hope that an earlier larger first level TLB had entries in it, which could still be used. Hence it is much simpler than the two-level TLB of the same size that we use for our base processor.

We compare our dynamic scheme with three base configurations that are identical in all respects, except for the data cache hierarchy. The first uses a two-level exclusive cache, with a direct mapped 256KB L1 cache backed by a 14-way 1.75MB L2 cache. 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. This configuration serves to ensure that the first base case does not perform poorly merely because of its exclusive nature. (An exclusive cache performs worse because every miss results in a swap or writebacks, which cause greater bus and memory port contention.) We also compare with a 64KB 2-way inclusive L1 and 2MB of 16-way L2, 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 all these caches, the L1 access is two cycles and the second level access is 15 cycles and is consistent with access times obtained from McFarland's model [14]. The conventional TLB is a two-level inclusive TLB with 64 entries in the first level and 448 entries in the second level.

4.2 Benchmarks

We have used a variety of benchmarks from SPEC95, SPEC2000, and the Olden suite [17]. These particular programs were picked 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. The chosen benchmarks were compiled with the Compaq cc, f77, and f90 compilers at an optimization level of O2. Warmup times were studied and the simulation was fast-forwarded through these phases (one of the Olden benchmarks (mst) is small enough that it can be simulated in its entirety). A further million instructions were simulated in detail to prime all structures before starting the performance measurements. The benchmarks are summarized in Table 2.

Benchmark	Suite	Datasets	Simulation window (instrs)
em3d	Olden	20,000 nodes, arity 20	1000M-1100M
health	Olden	4 levels, 1000 iters	80M-140M
mst	Olden	256 nodes	entire program 14M
perimeter	Olden	32Kx32K image	1428M-1528M
compress	SPEC95 INT	ref	1900M-2100M
hydro2d	SPEC95 FP	ref	200M-335M
apsi	SPEC95 FP	ref	200M-400M
swim	SPEC2000 FP	ref	1200M-1400M
art	SPEC2000 FP	ref	300M-500M

Table 2: Benchmarks

4.3 Timing Estimation

We use the cache and TLB timing model developed by McFarland [14] 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 scaling down to 0.1μm technology [15] (which we used for all delay calculations). 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 three conventional caches 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 [4]. Repeaters are used in the configurable cache as well as in the conventional L1 cache whenever they reduce wire propagation delay.

5 Performance Results

In this section, we compare the interval-based scheme with three base cases. The various configurations are tabulated in Table 3. We also ran the benchmarks with a perfect memory system (all data cache accesses serviced in one cycle) to estimate the contribution of the memory system to execution time. We refer to the difference in CPIs as the memory-CPI. Since our dynamic cache is only trying to improve memory performance, the memory-CPI quantifies the impact of our scheme on memory performance, while CPI quantifies the impact on overall performance.

Some of our benchmarks (the ones from the Olden suite) are much shorter than the SPEC programs. Hence, to get an overall metric of performance across all benchmarks, we use the HM of IPC. This would represent the performance of a workload where each application ran for an equal timeslice.

A	Base exclusive cache with 256KB 1-way L1 and 1.75MB 14-way L2
B	Base inclusive cache with 256KB 1-way L1 and 2MB 16-way L2
C	Base inclusive cache with 64KB 2-way L1 and 2MB 16-way L2
D	Interval-based dynamic scheme

Table 3: Summary of the various configurations

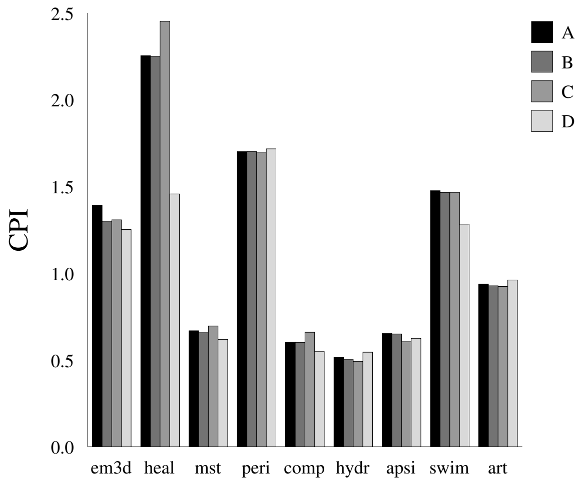

Figure 4: CPI for the 3 base cases (A, B, C) and the interval-based scheme (D)

Figures 4 and 5 show the CPI and memory-CPI for each of the applications. Table 4 summarizes the speedups with respect to each base case for every application. To quantify the effect of the TLB and the cache, we ran case A and case D with a perfect TLB to determine the improvement got with the cache alone, which is shown in the last column.

Benchmark	Speedup with respect to the 3 base cases			Speedup with only the cache
	64K-2-inc (C)	256K-1-inc (B)	256K-1-exc (A)	
em3d	1.05	1.04	1.11	1.08
health	1.68	1.55	1.55	1.19
mst	1.13	1.06	1.08	1.08
perimeter	0.99	0.99	0.99	0.99
compress	1.20	1.10	1.10	1.06
hydro2d	0.90	0.92	0.95	0.95
apsi	0.97	1.04	1.04	1.07
swim	1.14	1.14	1.15	1.06
art	0.96	0.97	0.98	0.98

Table 4: Summary of CPI speedups when compared to the three base cases and speedup with respect to case A, when assuming a perfect TLB for both

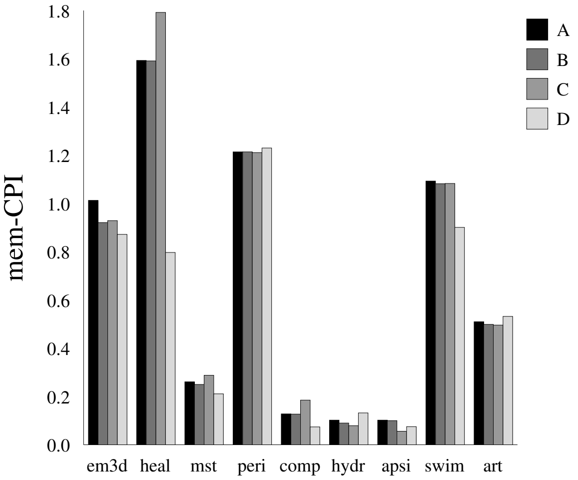

Figure 5: Mem-CPI for the 3 base cases (A, B, C) and the interval-based scheme (D)

When compared with base case A, the CPI speedup is as high as 1.55 for health, while the mem-CPI speedup is 2.0. The HMs of the IPCs for A, B, C and D are 0.882, 0.895, 0.874 and 0.999, which is a 1.14 speedup for D over A. The corresponding speedup for mem-IPC is 1.25. From the last column of Table 4, it is clear that in most cases the cache and TLB play commensurate roles in the improvement.

We can categorize our results based on the behavior of applications with working set sizes that either fit or do not fit in the available on-chip memory, as well as due to the effect of cache associativity.

Perimeter, em3d, swim, and art have a working set size larger than 2MB. Hence, perimeter remains at the minimum sized cache after an initial exploration phase and shows no change in performance (it sees a change of phase at a later point, but again picks the minimum sized cache). For Em3d, the best tradeoff point occurs at the 2MB cache size and the program remains stable at that size. Even though there is no sharp drop in miss rate at the 2MB cache size, there is a sharp increase in CPI because there is no (often fruitless) backup cache to be looked up. The saving of those 15 cycles on every trip to memory gives the CPI a boost. Swim also picks the 2MB cache size for the same reason. It does have 3 different execution phases, which cause it to become unstable and do an exploration before settling on the 2MB each time. Art is very unstable in its behavior and does not remain in any one phase for more than a few intervals. It 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. This results in a slight CPI degradation.

Health is an application with a working set size that fits in the available on-chip cache (2MB). The cache size of 1.5MB that the dynamic scheme stabilizes at helps bring down the miss intolerance drastically, resulting in a large CPI improvement. For mst, a cache size of 256KB provides the best trade-off point for most of the execution (the initialization phase of the mst (minimum spanning tree)). In the last few million cycles where the mst is computed, the working set size changes and the interval-based scheme increases the cache size in an effort to minimize the high miss intolerance. The program finishes execution before a stable state can be reached, however. Compress also dynamically adapts its cache configuration during execution and benefits from a larger cache (768KB). The program is fairly unstable and keeps changing phase every few intervals. The use of hit and miss intolerance help control the exploration phase and only 2 cache sizes are tried before settling on the 768KB cache.

Apsi gets its benefit from moving to a set associative cache. Even though our default cache is direct-mapped, the dynamic scheme has the ability to reconfigure and move to a larger set-associative cache, thereby showing a speedup of 1.04 with respect to the direct-mapped base cases (A and B). C performs slightly better (D has a slowdown of 0.97 compared to C) because its set-associative cache is smaller and hence faster. Note that apart from hydro2d and apsi, the 64KB 2-way cache (configuration C) always performs worse than base case B. Our dynamic scheme is able to reap the benefits of a set-associative cache for those applications with a large number of cache conflicts, while retaining the lower latencies and behavior of the direct-mapped cache for other applications. Like compress, apsi too is quite inconsistent in its behavior, resulting in frequent exploration. Again, the hit intolerance metric prevents extensive exploration.

Hydro2d has very inconsistent behavior across intervals and the algorithm hardly remains at stable state for more than one interval. Due to our choice of starting from the smallest cache size at the start of each unstable phase, only a small performance degradation occurs.

In terms of the effect of TLB reconfiguration, health, swim, and compress benefit from using a larger TLB. Health and compress require 256 and 128 entries for best performance, and the dynamic scheme is able to settle at this size. Swim shows phase change behavior with respect to TLB usage, resulting in 5 stable phases that require the entire range of TLB sizes. We notice a slight degradation in performance because of the configurable TLB in some of the benchmarks, because of the fact that the configurable TLB design is effectively a 1-level hierarchy using a smaller number of total TLB entries since data is not swapped between the primary and backup portions when handling TLB misses.

Em3d is the only benchmark where our choice of an exclusive cache severely degrades performance. Because of an L1 miss rate of more than 17%, the excess traffic between the first two levels severely degrades the exclusive cache performance compared to that of the inclusive caches. Hence, the speedup with respect to B and C is not as marked as that with respect to A. Note that em3d avoids the traffic between L1 and the backup ways by dynamically picking the 2MB cache size.

It must also be noted that the inclusive caches use the L2 as a unified cache, while the exclusive organizations have an L1 I-cache and memory at the next level. For the benchmarks studied, this did not affect performance, but performance degradation may occur for the configurable cache for programs with large instruction footprints.

6 Conclusions

We have described a novel configurable cache and TLB as an alternative to conventional two-level hierarchies. Repeater insertion is leveraged to enable dynamic cache configuration, with a cache organization that allows for dynamic speed/size tradeoff while limiting the impact of speed changes to within the memory hierarchy. Our configuration management algorithm is able to dynamically examine the tradeoff between hit and miss intolerance in hardware to determine appropriate cache size and speed. Our results show an average 14% with up to a 55% improvement in IPC in comparison with a conventional L1-L2 design of identical total size, with the benefit almost equally attributable on average to the configurable cache and TLB.

Future work includes exploiting the low-energy configurations and implementing energy-aware modifications to the configuration algorithm, exploring reconfiguring on subroutine granularity, and investigating the use of compiler support. For instance, for those subroutines where the cache size is a function of a dynamic runtime parameter, compiler support to instrument the subroutine so that this information can be utilized at runtime would be useful. In addition, for applications where a subroutine granularity is not the most appropriate, the compiler could choose appropriate adaptation points. We will explore these mechanisms for applications with regular access patterns, where it is likely to be most effective. The effect of similar mechanisms for the L2-L3 hierarchy also need to be studied. Finally, improvements at the circuit and microarchitectural levels will be pursued that better balance configuration flexibility with access time and energy consumption.

References

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

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

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

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

[5] 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.

[6] 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.

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

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

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

[10] 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.

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

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

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

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

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

[16] 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.

[17] 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.

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

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

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