2003 Workshop on Duplicating, Deconstructing and Debunking

June 8, 2002

San Diego, California

Organized by:
Bryan Black, Intel Labs, bryan.black@intel.com
Mikko Lipasti, University of Wisconsin, mikko@engr.wisc.edu

Final Program

Session 1: Memory Hierarchy

Session 2: System Software

Session 3: Advanced Microarchitecture

Cache-Conscious Allocation of Pointer-Based Data Structures Revisited with HW/SW Prefetching

Josefin Hallberg, Tuva Palm and Mats Brorsson

Department of Microelectronics and Information Technology (IMIT)
The Royal Institute of Technology (KTH)
SE-100 44 Stockholm, Sweden
{josan, tuva, matsbror}@kth.se

Abstract

As memory access times continue to be a bottleneck, differential research is required for better understanding of memory access performance. Studies of cache-conscious allocation and software prefetch have recently sparked research in the area of software optimizations on memory, as pointer-based data structures previously have been elusive to the optimizing techniques available. Research on hardware prefetch mechanisms have in some cases shown improvements, but less analytical schemes have tended to degrade performance for pointer-based data structures.

This paper combines four hardware schemes, normally not efficient on pointer-based data structures, and a greedy software prefetch with cache-conscious allocation to evaluate positive effects of increased locality, in a comparative evaluation, on five level 1 data cache line sizes.

We show that cache-conscious allocation utilizes large cache lines efficiently and that none of the prefetch strategies evaluated add significantly to the effect already achieved by the cache-conscious allocation on the hardware evaluated. The passive prefetching mechanism of using large cache lines with cache-conscious allocation is by far outstanding.

1 Introduction

As processor speeds are increasing and programs are becoming more memory intensive, memory access times are a bottleneck for performance. This situation is putting pressure on research for better data cache performance and some interesting efforts have recently been devoted to this area. Pointer-based data structures are usually randomly allocated in memory and will generally not achieve good locality, resulting in higher miss-rates. This has raised the need to handle the unpredictability of pointer-based data structures in an efficient way.

Two previously studied software-based strategies attempt to provide performance improvements specifically for applications using pointer-based data structures. The two techniques are software prefetch, [15, 16], and cache-conscious allocation of data, [6, 5, 7]. Those results showed that cache-conscious allocation is by far the most efficient optimization technique of the two. Software prefetch is, however, better suited for automatization and it has been efficiently implemented in a compiler to dynamically prefetch only hot data streams, [8], to limit the cost of the extra instructions. Cache-conscious allocation with a software prefetch scheme is evaluated in [2]. It compares the impact on bandwidth and verifies that latency and bandwidth trade off and limit the effectiveness of each optimization. It is concluded that software prefetch does not add significantly to the performance benefit of cache-conscious allocation.

Studies of hardware-based strategies have lately attempted, in some cases successfully, [11, 14, 19, 18, 23], to achieve performance improvements for pointer-based data structures, often referred to in these studies as linked data structures. These studies concentrate on calculating and prefetching pointers, [4, 11, 19, 23], pointer dependencies, [12, 18], and the effects of effectively predicting what to evict from the cache to accommodate prefetched data, [14], and they consequently require, more or less extra over-head, memory and/or instructions. The usefulness of general (e.g. next-line) hardware prefetch of pointer-based data structures is not encouraging, [21]. Strategies prefetching without knowledge of the data flow are likely to pollute the cache when applied to pointer-based data structures. However, these hardware strategies have the potential to take advantage of the increased locality of cache-consciously allocated data, [20].

In theory, prefetching and cache-conscious allocation should complement each other's weakness. Cache-conscious allocation should reduce the prefetch overhead of fetching blocks with partially unwanted data in the cache lines. Prefetching should reduce the cache misses and miss latencies between individual nodes of data structures in different cache-consciously allocated blocks. The difficulties

lie in achieving adequate correctness and precision. By combining the hardware prefetch with cache-conscious allocation on pointer-based data structures, the effects of both strategies can be completely exploited, without adding any instruction overhead of a software strategy.

The cache-conscious allocation and hardware prefetching strategies have never been merged and evaluated for performance and possible synergy effects. The software strategy is compared with four hardware strategies, normally inefficient on pointer-based data structures, and not requiring extra analysis, memory or instructions. The optimization strategies and the abbreviations used in this paper are found in Table 1. We will present a comparative evaluation of the strategies found in Table 2, and our conclusions of the gathered results.

Optimization	Abbrev.
No Optimizations	noopt
SW Prefetch	swpf
CC Allocation w cc-block 256	cc256
HW Prefetch on Miss	hwpfpom
HW Prefetch Tagged	hwpftagg
HW Prefetch on Miss, one cache block	hwpfoneblk
HW Prefetch on Miss, rest of cc-block	hwpfallblk

Table 1. Abbreviations of the optimization strategies

Sections 2 and 3, contain memory related performance characteristics and background of cache-conscious allocation and prefetching for pointer-based data structures.

	alone	with cc256
swpf	X	X
cc256	X	-
hwpfpom	X	X
hwpftagg	X	X
hwpfoneblk	-	X
hwpfallblk	-	X

Table 2. All combinations of the cache-conscious allocation and prefetching used in this study

The experimental framework is presented in section 4. To perform the experiments we have modeled a MIPS-like uniprocessor architecture in SimpleScalar, [3], and run four benchmarks of the Olden benchmark suite, [17]. We have analysed the performance effects of the techniques on five different cache line sizes. The performance evaluation of our results is found in section 5, and section 6 presents some of the related work in these areas. In section 7 are our conclusions and further issues to explore.

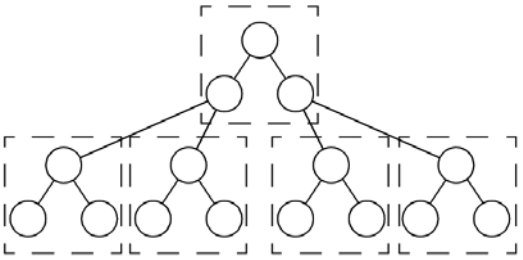


Figure 1. How nodes can be cache-consciously allocated in blocks to improve locality, (e.g. the next list node in a linked list or the children of a node in a tree)

2 Cache-Conscious Allocation

The technique of cache-conscious allocation is a technique worth further study as it has exhibited such excellent improvements in execution time performance. We have attempted to duplicate the cache-conscious allocation used by Chilimbi et al., [6]. Cache-conscious allocation can be adapted to the specific needs of a program by choosing the cache-conscious block size, or *cc-block size*, according to its data structures and to the specific cache line size of a system.

Cache-conscious allocation of data structures attempts to co-allocate data in the same cache line, so that cache line utilization is improved. By allocating data structures referenced after each other on the same cache line, better locality will be achieved, see Figure 1. This should lead to improved performance by a reduction of cache misses.

2.1 About ccmalloc

In this evaluation we have used a function called `ccmalloc()` for cache-conscious allocation of memory. The main difference from a regular `malloc()` is that `ccmalloc()` takes as an extra argument, a pointer to some data structure that is likely to be referenced close (in time) to the newly allocated structure. `ccmalloc()` attempts to allocate the new data in the same cc-block as the data structure pointed at by the argument pointer, as introduced in [5]. In the sample code in Figure 2 the parents and their children are attempted to be allocated together.

`ccmalloc()` invokes calls to the standard `malloc()` in two cases; when allocating a new cc-block or when the size of the data structure is larger than the cc-block. Otherwise, if called with a pointer to an already allocated structure, the new structure is put in empty slot in the cc-block right after that structure. When no proper area is found, ordinary `malloc()` is called with the cc-block size.

```
#ifdef CCMALLOC
  child = ccmalloc(sizeof(struct node),
   parent));
#else
  child = malloc(sizeof(struct node));
#endif
```

Figure 2. An example of how `ccmalloc()` is used to co-allocate a new node close to its parent node

2.2 Cache-Conscious Blocks, *cc-blocks*

The trade-off of cache-conscious allocation is that it demands cache lines large enough to contain more than one pointer structure in each, to improve hit rates and execution time. Thus the choice of the cc-block size is quite important. The bigger the blocks the lower the miss-rate if the allocation policy is successful, otherwise the memory overhead, i.e. fragmentation, can overwhelm other performance effects.

Previous studies on cache-conscious allocation used the same hardware cache line size as the cc-block size, [2, 5]. However, the cc-block size can be set dynamically in software, independently of the hardware cache line size. This means that even though the hardware cache line is smaller than the used data structures, `ccmalloc()` can take advantage of co-allocating data structures, and can be varied depending on the size of the data structures the programmer wants to co-allocate. In this study the cc-block size is set to 256 B, while the hardware cache line size is varied from 16 B to 256 B.

3 Prefetch

Prefetching structures before they are referenced will reduce the cost of a cache miss. Ideally the prefetching should start early enough so that the structure will be in the cache when referenced and thereby fully hiding the cache miss latency from the execution.

Prefetch can be controlled by software and/or hardware. Software prefetch results in extra instructions, which could affect performance by adding extra cycles to the execution time. Hardware prefetch does not lead to an instruction overhead, but to additional complexity in hardware. In our experiments the prefetching pertains only to the level 1 data cache.

3.1 Software Controlled Prefetch

Software prefetch is implemented by including a prefetch instruction in the instruction set. Prefetch instructions should be inserted in the program code, well ahead of reference, according to a prefetch algorithm. Several algorithms have been investigated in earlier studies on their own, [15, 16].

Pointer-based data structures often contain pointers to other structures, creating a chain of pointers. These pointers are dereferenced to find the prefetching addresses. The software controlled prefetch in this study is a greedy algorithm duplicated from Mowry et al., [15]. It is manually inserted in the code and does not require any extra memory or calculations. When a node is referenced, it prefetches all children of that node. This reduces cache miss latencies for the consecutively referenced children, as described in Figure 3. Without extra pointers or calculation, prefetching can only be done on the node's children, not its grandchildren.

Software prefetch is easier to control and optimize. As it only prefetches lines needed, the risk of polluting the cache with unused data decreases. The difficulty lies in getting the distance large enough to finish the prefetch before a reference. Software prefetch also imposes an instruction overhead caused by the prefetch instructions, possibly spoiling performance improvements gained by reduced cache miss latencies. It is also sensitive to bandwidth, [2], and issue width, [1].

3.2 Hardware Controlled Prefetch

There are several ways of implementing hardware prefetch support, [10, 13], and the algorithm choosing the lines to prefetch, [11, 19, 20, 22]. Depending on the algorithm used, prefetching can occur when a miss is caused or when a hint is given by the programmer through an instruction, or can always occur on certain types of data. The prefetch will fetch one or more extra lines into the cache.

We have implemented two hardware strategies originally described by Smith, [20], and later Vanderwiel, [21]: prefetch-on-miss, and tagged prefetch. The hardware prefetch mechanisms in this study attempt to utilize spatial locality, and do not analyze data access patterns. Pointer-based data structures usually do not respond well to these general strategies alone, due to their random allocation in memory and the difficulties to control the precision of the prefetches without extra analysis. We have also implemented two strategies that are designed to prefetch parts of the cache-consciously allocated blocks. These modified prefetch-on-miss strategies are implemented for the purpose of evaluating the other strategies' prefetch data overhead.

3.2.1 Prefetch-on-Miss

The prefetch-on-miss algorithm simply prefetches the next sequential line, *i+1*, when detecting a cache miss of line *i*. After handling a miss in the data cache a prefetch of the following line is always initialized. So each miss in the cache will lead to the fetch of two lines into the cache, if the line to prefetch is not already in the cache.

```
preorder(treeNode *t){
  if (t != NULL){
    prefetch(t->left);
    prefetch(t->right);
    process(t->data);
    preorder(t->left);
    preorder(t->right);
  }
}
```

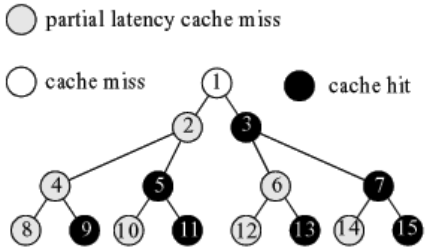


Figure 3. An example of how prefetch affects cache misses with the greedy algorithm, picture by [15]. The first node always gives a miss, if latency is two cycles. By prefetching both children the penalty will decrease for the following references.

The drawback of prefetch-on-miss is that it could lead to a lot of unused data in the cache, as it prefetches the next cache line on every miss. The performance of prefetch-on-miss is decided by the regularity of data references and their locality.

3.2.2 Tagged Prefetch

In the tagged prefetch strategy, each prefetched line is tagged with a prefetch tag. Like in the prefetch-on-miss strategy a cache miss of line *i* will lead to a prefetch of line *i+1*. When a prefetched line *i* is then referenced for the first time the tag is removed and line *i+1* is prefetched, though no miss has occurred.

This is an efficient prefetch method to use when memory is referenced fairly sequentially, and has been shown in studies without pointer-based data structures, [20, 21], to provide up to twice the performance improvements of prefetch-on-miss. However, as with prefetch-on-miss, prefetches are done indiscriminately on every miss and on referencing a prefetched line in the level 1 cache, risking unused data in the cache.

3.2.3 Prefetch-on-Miss, optimized for `ccmalloc()`

The hardware prefetch mechanism can be efficiently combined with cache-conscious allocation, by introducing a hint with the address to the beginning of such a block. We have implemented a detection mechanism that prefetches only cache-consciously allocated blocks. This mechanism is implemented with two different strategies, depending on how many cache lines to prefetch, prefetch-one-cc-on-miss, and prefetch-all-cc-on-miss.

Prefetch-one-cc-on-miss simply prefetches the next line after detecting a cache-miss on a cache-consciously allocated block, like the prefetch-on-miss but only on cc-blocks. The other, prefetch-all-cc-on-miss, decides dynamically how many lines to prefetch depending on where on the cc-block the missing cache line is located. This strategy prefetches all cache lines in the current cc-block from the address causing the miss and forward.

4 Experimental Framework

This section describes the hardware framework and the benchmarks on which the strategies were evaluated.

4.1 Hardware Architecture

The tests were conducted on an out-of-order, MIPS-like, uniprocessor simulator based on the SimpleScalar tool set, [3], with processor architecture parameters set according to Table 3. The memory latency is equivalent of 50 ns random access time, no wait states, for a 266 MHz bus, and a 3 GHz processor.

Prefetch handling was added to the simulator, introducing a prefetch instruction for the software prefetch, and hardware prefetch detection mechanisms for the hardware prefetch strategies. The benchmarks were compiled with the SimpleScalar GCC compiler for big-endian using the flags `'-lc -03'`.

4.2 The Benchmarks

The effects of merging cache-conscious allocation with either prefetch strategy were evaluated with applications from the Olden benchmark suite, [17]. Olden consists of ten applications with differing data structures and is commonly used to measure effects of architectural features.

We used four applications in our experiments, `health`, `mst`, `perimeter`, and `treeadd`. They were selected because they use dynamically allocated pointer-based data structures. Figure 4 shows their un-optimized stall times, indicating where the different benchmarks have their bottlenecks.

The busy time in Figure 4 seems to be extraordinarily low. However, since the processor model is an out-of-order

Architectural Parameter	Value
L1 D-Cache Size	16 kB
L1 D-Cache Line Size	{16 B, 32 B, 64 B, 128 B, 256 B}
L1 I-Cache Size	32 kB
L1 I-Cache Line Size	32 B
L1 Replacement Policy	Last Recently Used (LRU)
L1 Cache Associativity	2-way set-associative
L2 Unified Cache Size	512 kB
L2 Replacement Policy	LRU
L2 Cache Associative	2-way set-associative
D-TLB Size	512 kB
I-TLB Size	256 kB
L1 D-Cache Latency	2 cycles
L2 D-Cache Latency	12 cycles
Memory Latency	60 cycles(+ 10 cycles/sequential access)
Memory Access Bus Width	8 B
Load/Store Queue	8 entries
Instruction Fetch Queue	4 entries
Issue Width	4 instr/cycles
Functional Units	4 int, 4 FP, 2 memory
Multiplier/Divider	1 int, 1 FP
Branch Prediction Scheme	Bimodal
Branch Prediction Table Size	2048 B
Branch Target Buffer	4-way associative, 512 B
Branch Miss-Prediction Latency	3 cycles

Table 3. The Architectural Model

model, the concept of stall is not well defined. We use the same definition as has been done in many other previous studies: when the maximum number of instructions are retired in a clock cycle, that cycle is counted as busy. Otherwise, we say that the cycle is stalled due to the oldest instruction waiting to be retired. If that is a load- or store instruction, it is a memory stall, otherwise it is a FU stall. If there is no instruction waiting to be retired, the stall is said to be a fetch stall. This means that busy time is the fraction of all clock cycles when the full issue width can be utilized.

The optimization strategies are likely to have the greatest effect on the benchmarks where memory stalls are predominant. At the end of this section is an overview of benchmark parameters and behavior, see Table 4, chosen according to the studies that we are re-examining and combining.

`health` simulates a Columbian health care system. Elements are moved between lists during execution, and there is more calculation between data references compared to the other benchmarks. Because of poor data structures and algorithms, `health` is not an exemplary benchmark, pointed out by Zilles, [24]. As results from health are presented here, the reader is alerted to read those results with caution. They are still relevant for our memory allocation evaluations.

`mst` creates a graph and calculates its minimal spanning tree. The `mst` benchmark originally used a locality optimization procedure which made the effects of `ccmalloc()` non-existent. The data structures were allocated in 32 kB blocks, not fitting in the 256 B cc-blocks used in `ccmalloc()`. `mst` was thus modified to use an ordinary allocation procedure instead, to enable measuring the effects of `ccmalloc()`.

`perimeter` calculates the perimeter of a region of an image. The data structures are allocated in an order similar to access order, resulting in some kind of locality optimization. There are few calculations between references, complicating prefetch.

`treeadd` calculates a recursive sum of values in a balanced binary-tree. It is similar to `perimeter`, but has slightly more calculations between data references.

5 Performance Evaluation

In this section we present the performance evaluation. We begin with the impact on execution time. Then we present effects on cache performance and prefetch issues. Finally

Benchmark	Input Parameters	Primary Data Structure
Health	levels=5, max.time=500 probability=1	linked lists
Mst	nodes=512	array of linked lists
Perimeter	levels=12	quad tree
Treeadd	nodes=20	balanced binary tree

Table 4. The benchmarks from Olden Benchmark Suite used

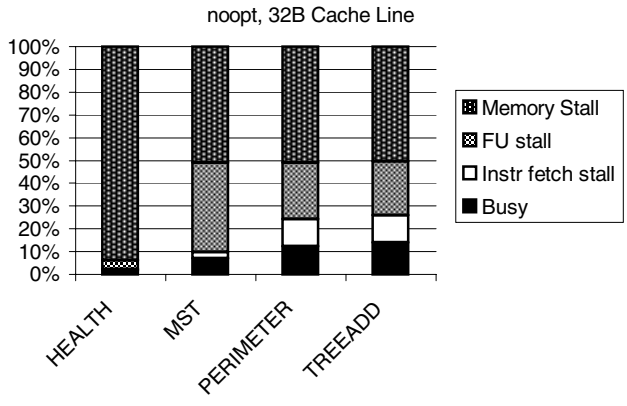


Figure 4. Stall times without cc-allocation or prefetching for the applications, on a 32 byte cache line

we discuss the memory and instruction overhead.

5.1 Execution Time

The execution times in Figure 5 show that cache-conscious allocation outperforms both hardware and software prefetch on their own, while software prefetch outperforms hardware prefetch without cache-conscious allocation of data. The data structures random location in memory makes sequential hardware prefetch volatile as there is no guarantee for the next sequential line ever being used, and as expected when there is no inherent locality in the data, the hardware strategies decrease performance for some simulations.

5.1.1 Effects of Cache Line Size

The combinations of prefetch strategies and cache-conscious allocation show that larger cache line sizes reduce the impact of prefetching. Large cache lines with cache-consciously allocated data decrease cache misses, and thus also the need and impact of prefetching. The improvements of the combined strategies are more noticeable on larger cache lines. By combining hardware prefetch with cache-conscious allocation, pollution, a common problem of large cache lines, decreases, due to improved locality.

5.1.2 Effects on Memory Stall

To evaluate the efficiency of our memory-targeted optimizations, stall times can show if memory stall is affected. These are presented for the 32 B line size, for the combinations of cache-conscious allocation, with software prefetch and with prefetch-on-miss, in Figure 6. The memory stalls for noopt are presented in section 5.2.

Stall times are reduced by 12% on average for the combined strategies. The software combination caused the greatest stall reduction for `health` and `mst`, and the hardware strategy is better for `perimeter` and `treeadd`.

5.1.3 Software vs. Hardware Prefetch, in combination with Cache-Conscious Allocation

In general the combinations of hardware prefetch with cache-conscious allocation outperform the combinations with software prefetch. The results show that the ability to exploit locality well is more important to improved performance than decreased miss latencies.

Software prefetch combined with cache-conscious allocation improves the results of software prefetch alone. However, it is less successful than cache-conscious allocation alone. The improved cache line utilization, decreased miss latencies, and successful prefetches, do not overcome the overhead caused by the prefetch instructions. The issue width of the hardware in this study, and data dependencies can limit the ability to schedule the prefetch instruction for early execution.

The results of hardware combinations with prefetch-on-miss and tagged hardware prefetch do not differ very much. The two cc-block aware strategies, prefetch-one-cc-on-miss and prefetch-all-cc-on-miss, do not outperform prefetch-on-miss, indicating that prefetch-on-miss exploits the locality of the cache-consciously allocated data well, without polluting the cache. The prefetch-all-cc-on-miss strategy behaves slightly worse than prefetch-one-cc-on-miss, indicating that

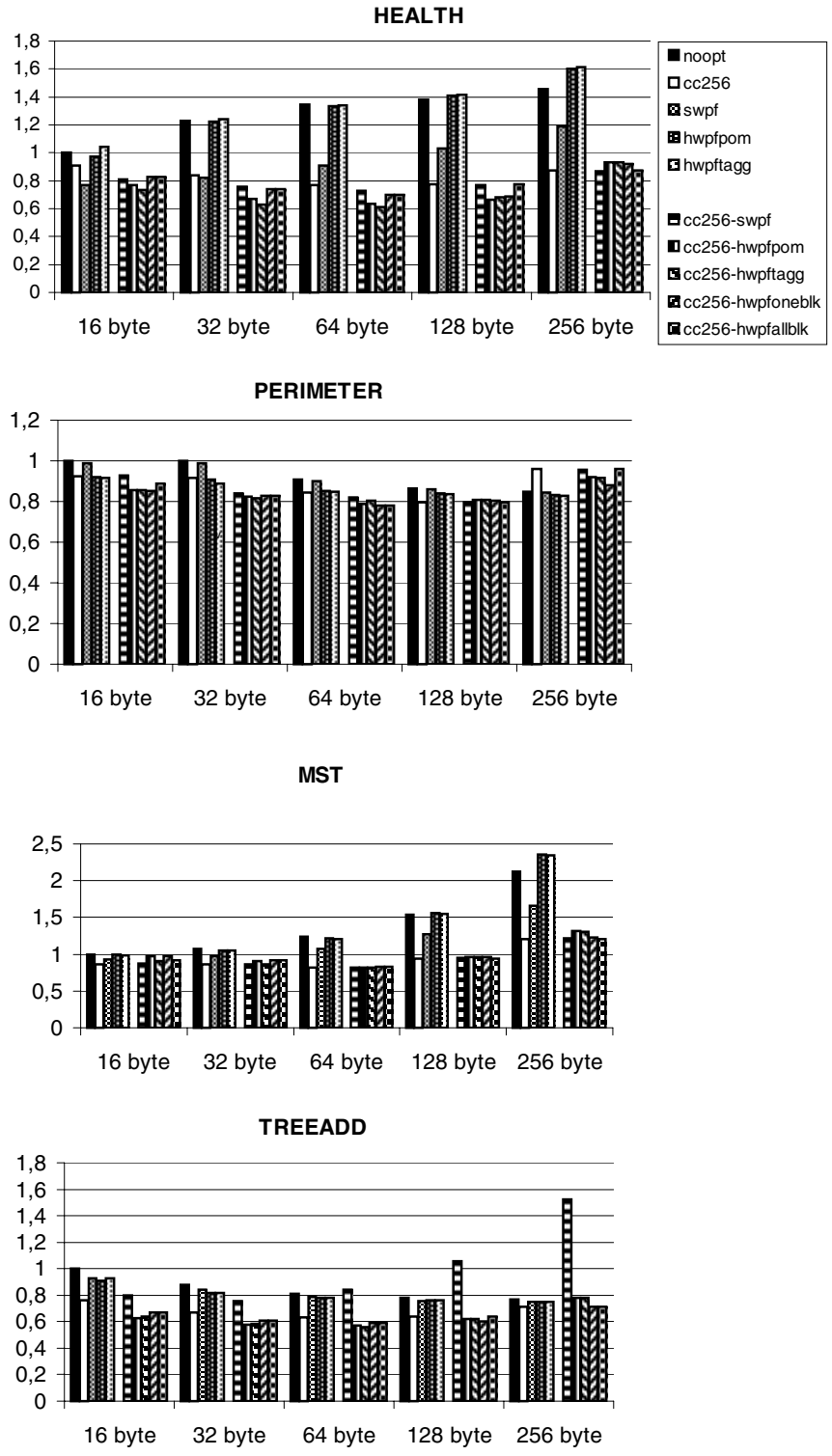


Figure 5. Normalized execution times for the applications, and all the various allocation and prefetch strategies for different cache block sizes.

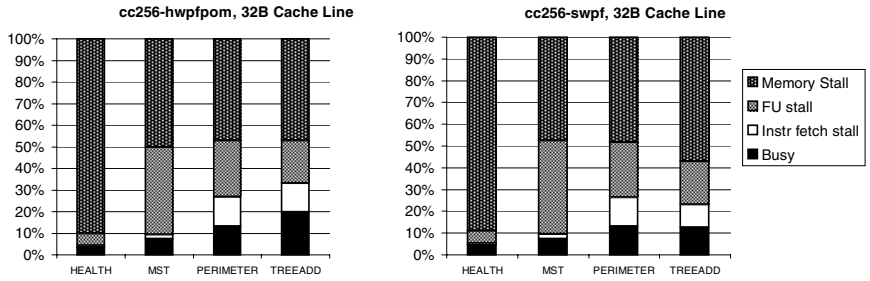


Figure 6. Stall times for cc-allocation combined with software prefetch (a) and hardware prefetch-on-miss (b)

prefetching all the data from the cc-block will cause conflict misses, throwing out data still in use.

5.2 Cache Performance

The miss-rates are improved by most optimization strategies, charts showing their improvements are found in Figure 7. The increased spatial locality with `ccmalloc()` reduces cache misses and minimizes cache pollution. Software prefetch generally shows a reduction in miss-rates. The combinations achieve the lowest rates, and the combination with software prefetch has the lowest miss-rates on average.

Figure 7 shows that large cache lines with cache-consciously allocated data are much more effective on cache misses than the implemented prefetchers. Hardware prefetch tends to prefetch too much unused data, and software prefetch tries to prefetch too much data that is already in the cache. Many of the prefetch instructions are thus unnecessary.

The fraction of loads that could be successfully prefetched, and partially hiding the latency, and unprefetched loads are found in Figure 8. It shows that the software prefetch achieves higher precision of prefetched data, resulting in more successfully prefetched data in the single strategy case.

Prefetch-on-miss and tagged prefetch, without cache-conscious allocation, do not result in a lot of successful prefetches at all, as shown in Figure 8. Prefetch-on-miss and tagged hardware prefetch increase miss-rates for small cache lines, but show a radical improvement for the largest cache line size. These results only imply that large cache lines are able sometimes to alleviate the bluntness of hardware prefetch even without locality.

When prefetching uses cache-conscious allocation there is a general increase of successful prefetches. The hardware strategies are more sensitive to cache line size than the software prefetch. Misses and tags trig the hardware prefetch, resulting in fewer attempts to prefetch data already in the cache. The hardware prefetch will, however, prefetch more unused data than software prefetch, as it lacks precision.

5.2.1 Software Prefetch combined with Cache-Conscious Allocation

Software prefetch combined with cache-conscious allocation results in an increased amount of used cache lines among the prefetched lines, shown in Figure 8. This is caused by the increased spatial locality allowing the accidental prefetch of a node that will be used that would otherwise cause a miss. However, it also results in an increased amount of prefetch instructions that tries to prefetch data already in the cache.

5.2.2 Hardware Prefetch combined with Cache-Conscious Allocation

The hardware strategies show greater improvements with the cache-conscious allocation than the combinations with software prefetch, Figure 7. Prefetch-on-miss and tagged prefetch do not differ very much in cache behavior.

The hardware strategies modified for cache-conscious allocation do not show any great results of prefetch though they provide more successful prefetch rates than prefetch-on-miss and tagged without cache-conscious allocation, shown in Figure 8. Further, the amount of unused but prefetched lines are larger than the amount of used prefetched lines when implementing prefetch in hardware without any detection due to problems with precision.

Although the amount of unused prefetched lines decreases when combining hardware prefetch with cache-conscious allocation, the amount is still high compared to software prefetch. The lack of precision renders hardware prefetch inefficient as the amount of unused data is high. The number of used prefetched lines decreases with larger cache lines. This is due to increased spatial locality, and to the reduced need of prefetch caused by the larger cache lines.

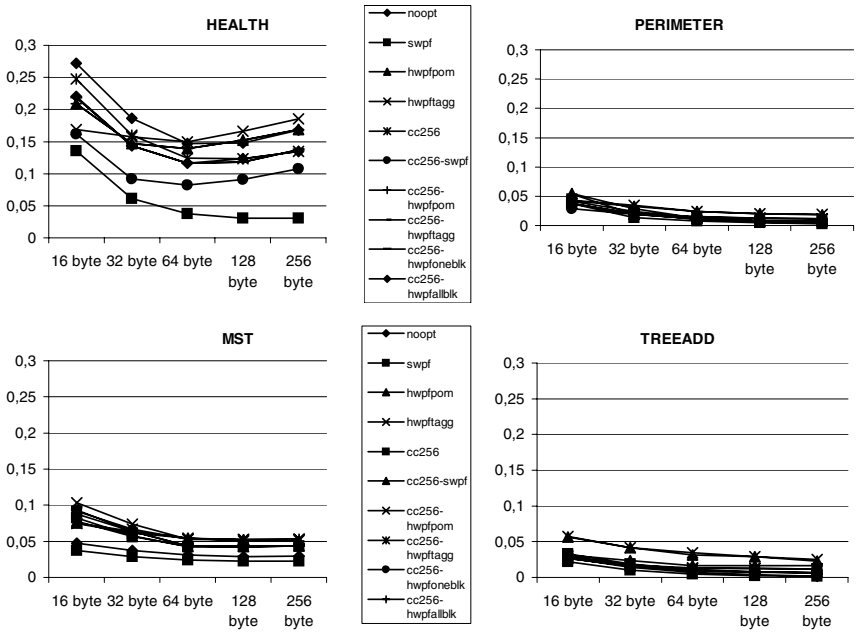


Figure 7. Level-1 data cache miss-rates for the applications and the various allocation and prefetch strategies

5.3 Memory and Instruction Overhead

Table 5 shows the allocated heap memory for all the benchmarks. For the prefetch strategies no extra memory is needed. `ccmalloc()`, however, uses more memory than the ordinary `malloc()`. This does not necessarily improve overall performance. This implies that the cc-block size has to be chosen carefully. Smaller cc-blocks require less memory, but when too small for the data structures allocation defaults to `malloc()`.

Software prefetch generates extra instructions, and the relative instruction increase is found in Table 6, for all the benchmarks. The positive effects of software prefetch are reduced and sometimes revoked by this overhead.

Health	Mst	Perimeter	Treeadd
20%	3.0%	0.57%	3.4%

Table 6. Software Prefetch Instruction Overhead, Relative Increase

6 Related Work

The research to improve performance for applications using pointer-based data structures has been restricted to cache-conscious layout manipulation and prefetch. To our knowledge this is the first evaluation of cache-conscious allocation combined and compared with both hardware and software prefetch.

In the field of prefetching, Mowry et al., [15, 16], have investigated three strategies for software prefetch of pointer-based data structures, using the Olden benchmarks and a simulated MIPS-like architecture. One of these, the greedy prefetch, is implemented in this study. Mowry et al. inserted the prefetch through a compiler, and we added the prefetch instructions manually. We managed to duplicate their results for `health`, `treeadd` and `perimeter`. For `mst`, however, their different allocation makes the effect of their software prefetch less prominent than ours.

Chilimbi et al., [6, 5, 7], have done extensive research on cache-conscious allocation, of which we have achieved similar results to Chilimbi's *new block* strategy , [6], which we have evaluated in the different combinations and on several cache line sizes. Chilimbi et al. also evaluated Mowry's greedy prefetch against `ccmalloc()` in [6]. The hardware prefetch in this study is not comparable to ours as it fetches loads and stores in the reorder buffer of 64 places. In a more recent study, Chilimbi et al. conclude that more profiling information seem necessary to prefetch with better results, [8], and that automatization of `ccmalloc()` is inefficient. Runtime systems with dynamic memory management are better suited for automating cache-conscious schemes, [9].

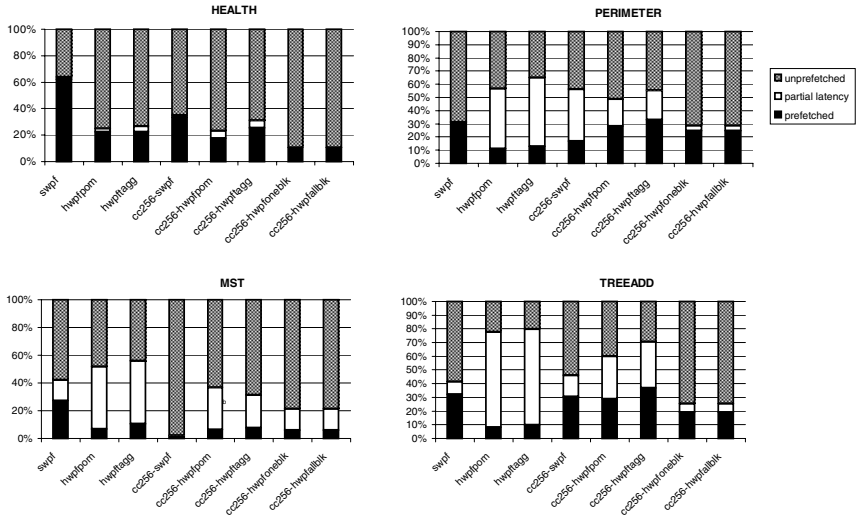


Figure 8. Prefetch Efficiency. The graphs show the fraction of loads that could be successfully prefetched, and the partially hiding, and the latency and unprefetched loads.

Allocation Strategy	Health	Mst	Perimeter	Treeadd
malloc	2756 kB	3596 kB	3080 kB	16488 kB
ccmalloc (cc-block 256B)	9336 kB	3876 kB	6188 kB	33980 kB

Table 5. Allocated Heap Memory for different allocation strategies

Badawy et al., [2], have evaluated the effects of combining software prefetch with cache-conscious allocation in benchmarks from the Olden benchmark suite, similar to our evaluation. Their software prefetch is, however, different from ours, adding jump-pointers in the data structures. They also have different hardware framework. Badawy et al. have evaluated the impact of different bandwidths, whereas we have evaluated the impact of different cache line sizes, in a uniprocessor system. According to Badawy et al., cache-conscious allocation only outperforms software prefetch when bandwidth is limited; with sufficient bandwidth software prefetch is the most successful strategy. However, their research also shows that the combination of cache-conscious allocation and software prefetch might not lead to further performance improvements, instead it counteracts changes in bandwidth or latency. Their results are similar to ours, although we have implemented a different software prefetch that does not require any extra memory.

Several researchers have studied hardware prefetch, or hybrid schemes, and successfully adapted hardware prefetch to pointer-based data structures with irregular access behavior. However, they generally require more hardware than those evaluated in this study. Hardware support has been investigated by the use of lock-up free prefetching, [13], and prefetch buffers, [10], and general prefetching in hardware is described in [20, 21] together with other cache memory aspects. Karlsson et al., [11], propose a technique for prefetching pointer-based data structures, either in software combined with hardware or in software alone, by implementing prefetch arrays, making it possible to prefetch both short data structures and longer data structures without knowing the traversal path. Roth et al. have investigated more adaptable strategies for hybrid prefetch schemes, using dependence graphs, [18], and jump pointer prefetching, [19]. In [19], Roth et al. evaluate a framework for jump-pointers implemented in turn in software, hardware, and in a hybrid scheme, in which the hybrid scheme outperforms each scheme on its own.

Annavaram et al., [1], have performed research of both the instruction overhead and lack of spatial locality, and how they are affected by increased issue widths. Their research shows that out-of-order processors with a wide issue width can hide memory latency, making pointer prefetch less useful, and that as the issue width increases, the lack of spatial locality tends to cause performance degradation.

7 Conclusions

Cache-conscious allocation seems to be an efficient way to overcome the drawbacks of large cache lines. This is due to the passive hardware prefetch of cache-conscious allocation. The combinations of all prefetch strategies and cache-conscious allocation show that the larger the cache line size the less impact of prefetch. As the cache line gets larger, the positive effects of prefetch are less prominent compared to the use of cache-conscious allocation alone. With large cache lines and cache-consciously allocated data, the cache misses decrease, and thereby both the need and impact of prefetching decrease.

Combining cache-conscious allocation with hardware prefetch can be unnecessary, as it seems that the effect of cache-conscious allocation alone is not outdone by any combination. However, cache-conscious allocation can be used to overcome negative impact of next-line hardware prefetch on applications using pointer-based data structures. Our study further shows that hardware prefetch is better at exploiting cache-conscious data than software prefetch, in the hardware used. With a larger issue width these results may change.

The successful hardware prefetch strategies generally require extra memory and analysis, which can be compared to the memory required by cache-conscious allocation. This overhead is also partly true of our prefetch schemes, but not for those, that, in combination with cache-conscious allocation, give the best results. One conclusion of the gathered results from previous studies and ours is that when a compiler can use profiling information to optimize memory allocation in a cache-conscious fashion, the effort required for the hardware prefetch engine is limited. However, when profiling is too expensive performance will likely benefit from elaborate prefetch support.

Further studies in this area can include comparisons with more elaborate hardware and hybrid prefetching schemes to exploit cache-conscious allocation, and varying issue width as well as bandwidth in the hardware. Even if the possibilities of automating `ccmalloc()` are limited, as it requires extensive analysis of data flow, the use in environments where more cache-consciousness is available with garbage collection should not be overlooked. It would also be interesting to study how well hardware support can be applied to object-oriented programs and be used by virtual machines wanting to optimize cache-consciousness.

8 Acknowledgements

The authors would like to thank Todd Mowry at Carnegie-Mellon University, for providing the source code for the Olden benchmarks that we have run, and Youtao Zang of Arizona University, for providing the base code for `ccmalloc()`. The authors are also grateful to the anonymous reviewers for their helpful comments.

9 References

[1] Murali Annavaram, Gary S. Tyson, and Edward S. Davidson. Instruction overhead and data locality effects in superscalar processors. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 95–100, April 2000.

[2] Abdel-Hameed A. Badawy, Aneesh Aggarwal, Donald Yeung, and Chau-Wen Tseng. Evaluating the impact of memory system performance on software prefetching and locality optimizations. In *Proceedings of the 15th ACM International Conference on Supercomputing (ICS-01)*, pages 486–500, June 2001.

[3] Doug Burger and Todd M. Austin. *The SimpleScalar Tool Set, Version 2.0*. info@simplescalar.com.

[4] Shimin Chen, Phillip B. Gibbons, Todd C. Mowry, and Gary Valentin. Fractal prefetching b-trees: optimizing both cache and disk performance. In *Proceedings of 2002 ACM SIGMOD International Conference on the Management of Data*, pages 157–168, 2002.

[5] Trishul M Chilimbi, Bob Davidsson, and James R. Larus. Cache-conscious structure definition. In *Proceedings of Conference on Programming Languages Design and Implementation '99 (PLDI)*. ACM, SIGPLAN, May 1999.

[6] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 1999.

[7] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Making pointer-based data structures cache conscious. *IEEE Computer*, 33:12:67–74, December 2000.

[8] Trishul M. Chilimbi and Martin Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of Conference on Programming Languages Design and Implementation '02 (PLDI)*. ACM, SIGPLAN, May 2002.

[9] Trishul M. Chilimbi and James R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *Proceedings of the first international symposium on Memory management*, pages 37–48. ACM Press, 1998.

[10] Norman P. Jouppi Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373. IEEE, June 1990.

[11] Magnus Karlsson, Fredrik Dahlgren, and Per Stenstrm. A prefetching technique for irregular accesses to linked data structures. In *Sixth International Symposium on High-Performance Computer Architecture (HPCA-6)*, pages 206–217, January 2000.

[12] Nicholas Kohout, Seungryul Choi, Dongkeun Kim, and Donald Yeung. Multi-chain prefetching: Effective exploitation of inter-chain memory parallelism for pointer-chasing codes. In *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques*, Barcelona, Spain. September 2001.

[13] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the Eighth International Symposium on Computer Architecture*, pages 81–87. ACM, SIGARCH, May 1981.

[14] An-Chow Lai, Cem Fide, and Babak Falsafi. Dead-block prediction and dead-block correlating prefetchers. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 144–154, 2001.

[15] Chi-Keung Luk and Todd C. Mowry. Automatic compiler-inserted prefetching for pointer-based applications. *IEEE Transactions on Computers*, 48:2:134–141, 1999.

[16] Chi-Keung Luk and Todd C. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of 7th Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, October 1996.

[17] Olden benchmark suite v. 1.01, June 1996.

[18] Amir Roth, Andreas Moshovos, and Gurindar S. Sohi. Dependence based prefetching for linked data structures. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ACM Press)*, pages 115–126, 1998.

[19] Amir Roth and Gurindar S. Sohi. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 111–121, 1999.

[20] Alan J. Smith. Cache memories. *ACM Computing Surveys*, 14:3:473–530, September 1982.

[21] Steven P. VanderWiel and David Lilja. Data prefetch mechanisms. *ACM Computing Surveys*, 32:2:174–199, June 2000.

[22] Chengqiang Zhang and Sally A. McKee. Hardware-only stream prefetching and dynamic access ordering. In *International Conference on Supercomputing*, pages 167–175, 2000.

[23] L. Zhang, S. McKee, W. Hsieh, and J. Carter. Pointer-based prefetching within the impulse adaptable memory controller: Initial results. In *Proceedings of the Workshop on Solving the Memory Wall Problem*, June 2000.

[24] Craig B. Zilles. Benchmark health considered harmful. *Computer Architecture News*, 29:3, 2001.

Comparison of State-Preserving vs. Non-State-Preserving Leakage Control in Caches

Dharmesh Parikh‡, Yan Zhang†, Karthik Sankaranarayanan‡, Kevin Skadron‡, Mircea Stan†
†Dept. of Electrical and Computer Engineering, ‡Dept. of Computer Science
University of Virginia
Charlottesville, VA 22904
{*dharmesh,karthick,skadron*}*@cs.virginia,edu,* {*yz3w,mircea*}*@virginia.edu*

Abstract

This paper compares the effectiveness of state-preserving and non-state-preserving techniques for leakage control in caches by comparing drowsy cache *and* gated-V_{ss} *for data caches using 70nm technology parameters. To perform the comparison, we use "HotLeakage", a new architectural model for subthreshold and gate leakage that explicitly models the effects of temperature, voltage, and parameter variations, and has the ability to recalculate leakage currents dynamically as temperature and voltage change at runtime due to operating conditions, DVS techniques, etc.*

By comparing drowsy-cache and gated-V_{ss} at different L2 latencies, we are able to identify a range of operating parameters at which gated-V_{ss} is more energy efficient than drowsy-cache, even though gated-V_{ss} does not preserve data in cache lines that have been deactivated. We are also able to show potential further benefits of gated-V_{ss} if an effective dynamic adaptation technique can be found.

This paper duplicates some of the findings of both the drowsy-cache and "cache-decay" papers, but also debunks a fairly widespread belief that state-preserving techniques are inherently superior to non-state-preserving techniques.

1 Introduction

Power is rapidly become a design constraint not only in the domain of mobile devices but also in high performance processors. Although dynamic power —caused by switching activity—is the major source of total power dissipation in today's process generation, static power—caused by leakage current even when circuits are not switching—is gaining in importance for CMOS designs due to technology scaling. The 2001 International Technology Roadmap for Semiconductors (ITRS) [27] predicts that by the 70nm generation, leakage may constitute as much as 50% of total power dissipation. This makes efforts at leakage control essential to maintain control of power dissipation in both high-performance and mobile/embedded processors.

Recently, a great deal of research work in the architecture community has focused on reducing leakage power in the caches [11, 14, 15, 19, 25, 31, 33], branch predictor [16, 17], register file [2], issue queues [7, 8, 12, 24], and the ALUs [10]. Leakage control at the architecture level is attractive, because architectural techniques can control large groups of circuits (*e.g.* cache lines, banks, or the entire cache) at once. Leakage control for caches has been an especially active area of study because caches comprise such a large portion of chip area. Recent work [11, 14] has suggested that *state-preserving* techniques are the best choice for leakage control in the first-level (L1) caches, because they do not incur costly accesses to the second-level (L2) cache when reading data that has been placed in low-leakage or "standby" mode.

This paper shows that when the L2 cache offers a sufficiently fast access time (*e.g.*, when the L2 is on chip), *non*-state-preserving techniques can be superior. And even when the L2 is not especially fast, non-state-preserving techniques can still be superior if runtime adaptivity can identify the proper decay interval.

To perform this study, we use *HotLeakage* [32], a new architectural model for subthreshold and gate leakage that has been publicly released on the web. HotLeakage explicitly models the effects of temperature, voltage, and parameter variations, and has the ability to recalculate leakage currents dynamically as temperature and voltage change at runtime due to operating conditions, DVS techniques, etc.

The next section of this paper describes the two leakage-control techniques that we study and the timing and performance assumptions that we make in our simulations, and then Section 3 provides an overview of the HotLeakage model. Section 4 describes the rest of our simulation setup and the benchmarks we use, Section 5 presents the results of our comparison study, and Section 6 concludes the paper.

2 Leakage Control for Caches

The design space for low-leakage caches is daunting, encompassing the choice of size and threshold voltage for each transistor, the row and bitline length, and many more parameters too numerous to mention. Here we focus on just one dimension that can be treated within the scope of a single paper, namely the choice of state-preserving versus non-state-preserving architectural leakage-control techniques in the L1 data cache.

Recent literature has suggested that state-preserving techniques are preferable for leakage control in L1 D-caches, because they do not lose data values and hence do not unnecessarily incur the extra delay and energy associated with reloading that data from the L2 cache. In contrast, our results suggest that this is often not so, that the extra cost of accessing L2 with non-state-preserving techniques is offset by other important factors.

Hanson et al. [14] found that for L1 caches, *reverse body bias* (RBB) or auto-backgate-controlled MTCMOS (ABB-MTCMOS) [23]—a state-preserving technique that manipulates threshold voltages—outperformed gated-V_{ss}, primarily because they used long decay intervals that minimized opportunities for saving energy, and because they did not decay the cache tags [13] (thus avoiding time wasted to waken and read the tags on misses). We have chosen not to study RBB here, both because RBB presents some manufacturing challenges and, more importantly, because recent work by Intel suggests that its effectiveness is limited at future technology nodes by gate-induced drain leakage (GIDL).

Flautner et al. [11] did not directly compare their proposed drowsy-cache scheme against gated-V_{ss}, but suggested that its state-preserving nature is a major advantage.

2.1 Lowering the Quiescent V_{DD} (Gated-V_{ss})

Leakage currents decrease as the supply voltage (V_{dd}) is lowered. The *gated-V_{dd}* structure was introduced as a micro-architecture technique by Powell et al. in [25] as a way to reduce leakage power by using a high threshold "header" transistor to disconnect a cell, row, or way in the cache from V_{dd}. This high-threshold transistor drastically reduces the leakage of the circuit because it breaks the connection to the power supply. While this technique is efficient in saving leakage, there is the disadvantage that the cell loses its state (information). This means that there will be some performance penalty when the data in the cell is accessed and needs to be fetched from a farther level of the cache. This is harmless if the next access to that line would have been an eviction anyway (*true miss*); but if useful data was discarded, the next access will be an *induced miss.* This has important consequences. First and foremost it causes dynamic power dissipation due to an extra L2 access. Second, an induced miss might cause the program to run longer and hence increase total energy consumption. Gated-V_{dd} was proposed in [19] for shutting down individual lines in a cache to save leakage when a line is idle. Because the sleep transistor is more effective as a "footer" on the connection to ground–it is easier to prevent bitline leakage this way–this technique is better called *gated-V_{ss}*.

2.2 Drowsy Caches

An alternative method, proposed by Flautner et al. in [11], achieves significant leakage reduction by putting a cache line into a low-power drowsy mode. In drowsy mode, the information in the cache line is preserved by switching its V_{dd} to a separate power supply that is only about 1.5 times the threshold voltage. This reduces leakage current dramatically due to short-channel effects and preserves the value that is stored, making this another *state-preserving* technique. Like MTCMOS, there is still some overhead because V_{dd} must be returned to the proper level before the value can be safely read. Drowsy caches do not reduce leakage as much as *gated-V_{ss}*, because the cells are not fully disconnected from the power supply. The advantage of drowsy cache is the low penalty of accessing a drowsy line in standby: induced misses do not require an L2 access but only 1-2 cycles to restore the full voltage for that line. Induced misses for drowsy caches might therefore better be called *slow hits*.

2.3 Modeling of Cache Leakage Control

We have implemented a generic abstraction for modeling leakage control techniques based on putting individual lines into standby mode, allowing us to study techniques like gated-V_{ss} [19], drowsy cache [11], and reverse-body-bias [23].

Most dynamic leakage-control techniques partition a structure into active and passive portions. This can be done at various granularities; most recent work has done this at the granularity of rows in the SRAM array, which correspond to cache lines.

These leakage control techniques also require some extra hardware that adds to the area of the structure. Hence, these methods have the following costs:

1. Dynamic power due to the extra hardware
2. Leakage power due to the extra hardware
3. Dynamic power due to mode transitions (active to standby and vice-versa)
4. Dynamic power due to extra execution time, resulting either from extra latency in accessing the structure or extra latency in fetching data from the L2 cache.

The energy benefit of the techniques we have described is the leakage power saved in the lines that are in standby mode. This saving is proportional to the average percent area that is kept in standby mode (the *turnoff ratio*). Our experiments compute a *net* energy savings that subtracts from this gross benefit the costs itemized above: Wattch automatically capatures the extra energy due to longer runtime (item #4 above); this is compared to the energy from a baseline simulation with no leakage control, and the resulting cost is added to the other costs itemized above (#1–3). These are then subtracted from the gross leakage savings.

For both techniques, we use a global counter that counts from zero up to one-fourth the decay interval (defined as *update window size* in [11]) and then starts over. Following [19], each line uses a local two-bit counter; when the global counter reaches its maximum value, all two bit counters are incremented. When a two-bit counter reaches its maximum, the line has been idle for the full decay interval, it is assumed that the line's usefulness has decayed, and the line is deactivated. In the original drowsy-cache paper, this corresponds to the *noaccess* policy. Drowsy cache also proposes the *simple* policy, which uses no per-line access history but rather automatically turns off all lines every N cycles. The simple policy loses out in performance compared to the noaccess policy, but saves more leakage power. The difference seems modest for drowsy due to the fairly low cost of any extra slow hits: there is some increase in performance loss, but also more energy savings. To be fair to both gated-V_{ss} and drowsy, we used the same policy involving counters, namely noaccess.

For both techniques, we decay the tags too (defined as *drowsy tags* in [11]). Access to a drowsy line in such a case takes at least three cycles due to the need to wake up tags before they can be checked. For gated-V_{ss}, on the other hand, a line in standby mode has no useful information, and tags need not (cannot) be checked. This means that on a true miss to L2 when tags are in standby, gated-V_{ss} is actually faster. Hanson et al. also kept the tags awake in their study [13, 14].

A few other simulation details are worth mentioning. The time taken for a line to go to a low-leakage mode from high-leak normal mode (*settling time*) and vice versa was found from circuit simulation and is given in Table 1. Also, for both leakage saving techniques we use the same values of threshold voltage for all

	Drowsy	Gated-V_{ss}
Low leak mode to high	3	3
High leak to low	3	30

Table 1. Settling Time.

the transistors of the same type for a memory cell. In contrast, drowsy uses high-V_t for the access transistors. Modeling this is easy with HotLeakage. But for making fair comparison, we use the same threshold voltages (for 70nm we use 0.190 V for N-type and 0.213 V for P-type transistor). It is true that high-V_t access transistors help drowsy more than gated-V_{ss}. High-V_t access transistors only help gated-V_{ss} when lines are awake, while they help drowsy in both situations. But since the bulk of the leakage is when awake, we felt that using the same V_t was the best solution. Finally, HotLeakage models inter-die variation. We use the following three-sigma values for 70nm technology. The values were obtained from [22].

- Length of the transistor: 47%
- Thickness of the gate oxide: 16%
- Supply voltage: 10%
- Threshold Voltage: 13%

The simulator currently models leakage control in caches using the above costs and benefits. The dynamic power calculations are performed using Wattch routines—see Section 4 for details. The leakage power is calculated using our model as configured by the command line options—see Section 3 for details.

3 An Accurate Leakage Model for Architects

Although architectural control of leakage energy has been an active area of research in recent years, many of these studies use only abstract models of leakage that do not fully account for all effects that may impact leakage, like supply voltage and temperature; and other studies use circuit-extracted parameters that are not easily incorporated into other researchers' models. Unlike for dynamic power, where widely-available simulators like Wattch [5] have enabled a widespread body of research, there is no widely available model for leakage power. This inhibits leakage research and leads to approximate experiments. Although Butts and Sohi [6] propose a simple model for use at the architecture-simulation level of abstraction, no corresponding software is available. Most importantly, their model cannot easily model leakage when temperature, supply voltage, or threshold voltage vary dynamically: a new "normalized leakage" and k_{design} must be calculated for every possible value. This is inconvenient although feasible for leakage-control schemes like drowsy cache that uses two supply voltages, but intractable for any leakage studies that account for dynamically varying temperature or involve dynamic voltage scaling. Unlike the Butts and Sohi model, we find that k_{design} does in fact vary with temperature, supply voltage, threshold voltage, and channel length. Detailed plots can be found in [32].

We have developed and released a software model of leakage—based on BSIM3 [3] technology data and the Butts and Sohi abstractions—that is computationally very simple, can easily be integrated into popular power-performance simulators like Wattch, can easily be extended to accommodate other technology models, and can easily be used to model leakage in a variety of structures (not just caches, which are the focus of this paper). We call our model HotLeakage, because it includes the exponential effects of temperature on leakage. Temperature effects are important, because leakage current depends exponentially on temperature, and future operating temperatures may exceed 100° C [27]. In fact, HotLeakage also includes the heretofore unmodeled effects of supply voltage, gate leakage, and parameter variations.

HotLeakage has high accuracy because parameters are derived from transistor-level simulation (Cadence tools). Yet like the Butts and Sohi model, simplicity is maintained by deriving the necessary circuit-level model for individual cells, like memory cells or decoder circuits, and then taking advantage of the regularity of major structures to expresse leakage in simple formulas similar to the Butts-Sohi model. All necessary components of this formula are encapsulated in lookup tables.

We hope that this new leakage model and its public availability will facilitate greater research on techniques for controlling leakage power at the architecture level. HotLeakage is publicly available for download at `http://lava.cs.virginia.edu/HotLeakage`. It is a separate library with minimal dependence on the details of SimpleScalar and Wattch, so porting HotLeakage for use with other simulators should be straightforward. We encourage not only such ports, but also any other modifications or extensions users might wish to add.

3.1 Subthreshold Leakage

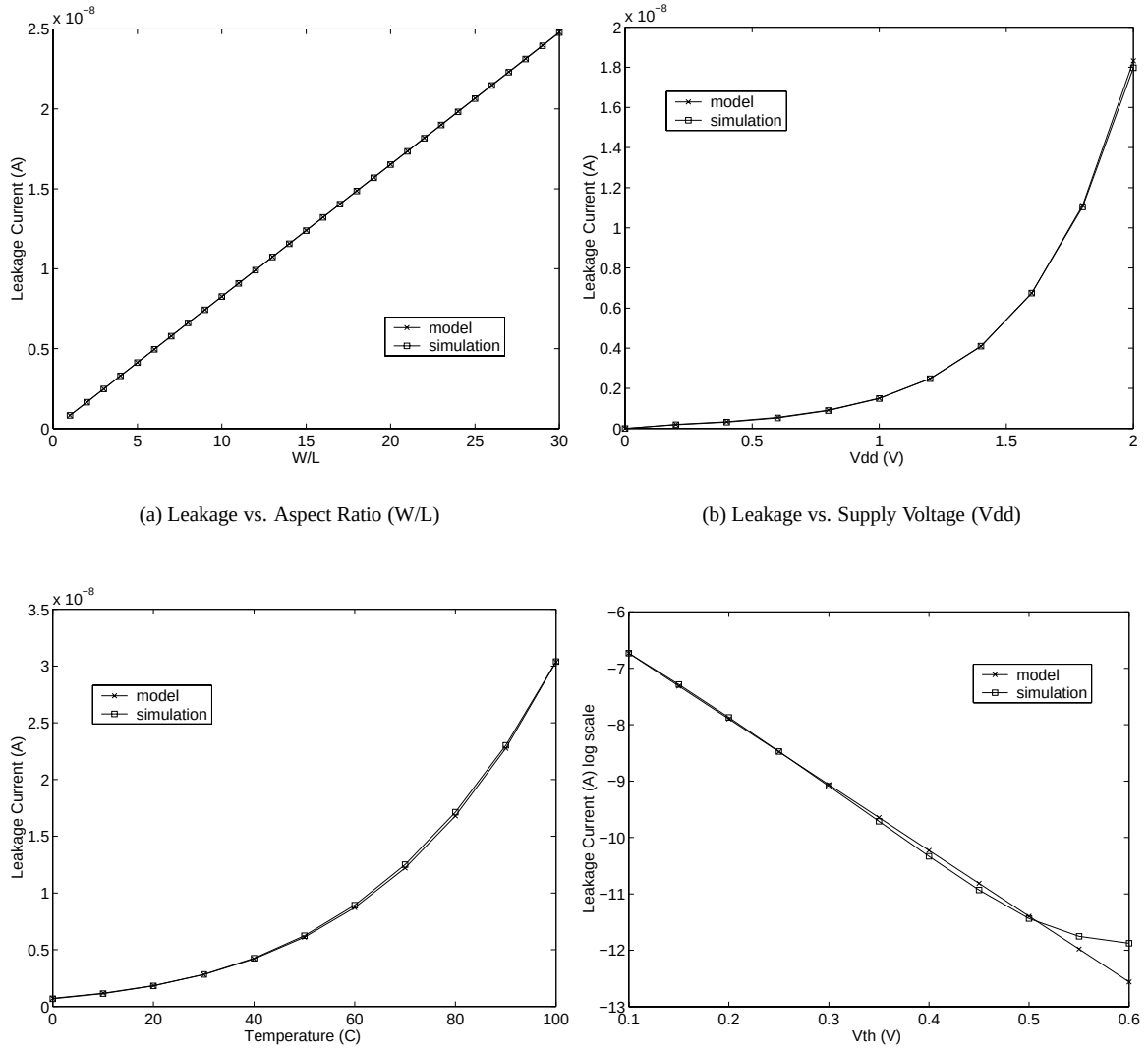
Leakage current is influenced by the threshold voltage, channel physical dimensions, channel/surface doping profile, drain/source junction depth, gate-oxide thickness, V_{dd}, temperature, and variations in all these parameters. For long-channel devices, the leakage current is dominated by leakage from the drain-well and well-substrate reverse-bias pn junctions. For short-channel transistors, because of the low threshold voltage, sub-threshold leakage is much higher. As gate oxides continue to scale, gate leakage is also becoming important. Keshavarzi, Roy, and Hawkins give an overview of these different leakage mechanisms in [20].

Our techniques for modeling gate leakage and parameter variations are described in Sections 3.2 and 3.3. Our technique for modeling sub-threshold leakage and its dependence on temperature, etc. is to extend the high-level model of sub-threshold leakage proposed by Butts and Sohi [6]. Their model neatly compartmentalizes some different issues affecting static power in a way that makes it easy to reason about leakage effects at the micro-architecture level. Leakage is given by the following equation:

$$P_{static} = V_{CC} \cdot N \cdot k_{design} \cdot \hat{I}_{leak} \quad (1)$$

This formula must be computed for each circuit or block of interest, *e.g.* the data array or a cache or the cache's "edge logic" (decoders and sense amplifiers). V_{CC} is the supply voltage, and N is the number of transistors in the circuit, which could be estimated by comparing it with a circuit of known functionality. k_{design} is a factor determined by the specific circuit topology and accounts for effects like transistor sizing, transistor stacking and the number and relationship of NMOS and PMOS transistors in a circuit. $\hat{I}_{leak}$ is a normalized leakage value for a single transistor, which we refer to as *unit leakage*.

Using this model, it is easy to see the relationships of some major factors that a processor designer can control for leakage-power



(a) Leakage vs. Aspect Ratio (W/L)

(b) Leakage vs. Supply Voltage (Vdd)

(c) Leakage vs. Temperature (T)

(d) Leakage vs. Threshold Voltage (Vth)

Figure 1. Comparisons of the proposed HotLeakage model against circuit-level simulation results.

savings: given a unit leakage $\hat{I}_{leak}$, leakage power is proportional to operating voltage and the number of transistors in the unit of interest. For example, DVS affects leakage by reducing V_{CC}, and "turning off" some unit (a cache bank or part of an issue queue) by disconnecting its power supply effectively reduces N.

In the Butts and Sohi formulation, the unit leakage $\hat{I}_{leak}$ is calculated once and assumes fixed values for threshold voltage (V_T), operating temperature, etc. Since recent work in low-leakage cache design [11, 14, 23, 26] as well as broader processor-design issues like thermal management [4, 18, 28, 29] manipulate parameters like V_T and temperature that are hidden in k_{design} or $\hat{I}_{leak}$, computing one fixed value for k_{design} and $\hat{I}_{leak}$ is not well-suited for actual simulation work (see [32] for more details).

To develop a portable simulation module for use with various architecture-level simulators, we retain the notions of k_{design} and unit leakage but compute the unit leakage dynamically during the simulation using the BSIM3 [3] leakage-current equation. This lets us explicitly account for temperature, supply voltage, and threshold voltage as key parameters, and includes the important DIBL effect which is sensitive to supply voltage. We also use two separate k_{design}'s for P- and N-type.

3.1.1 Unit Leakage

Based on the BSIM3 v3.2 [3] equation for leakage in a MOSFET transistor, our leakage model of a single transistor is given by the following equation:

$$I_{leakage} = \mu_0 \cdot C_{OX} \cdot \frac{W}{L} \cdot e^{b(V_{dd}-V_{dd0})} \cdot v_t^2 \cdot \left(1 - e^{\frac{-V_{dd}}{v_t}}\right) \cdot e^{\frac{-|v_{th}|-V_{off}}{n \cdot v_t}} \quad (2)$$

Low-level parameters are derived using transistor-level simulations: u_0 is the zero bias mobility, C_{OX} is gate oxide capacitance per unit area, W/L is the aspect ratio of the transistor, $e^{b(V_{dd}-V_{dd0})}$ is the DIBL factor derived from the curve fitting method, V_{dd0} is the default supply voltage for each technology (V_{dd0}=2.0 for 180nm, V_{dd0}=1.5 for 130nm, V_{dd0}=1.2 for 100nm and V_{dd0}=1.0 for 70nm), $v_t = kT/q$ is the thermal voltage, V_{th} is threshold voltage which is also a function of temperature, n is the subthreshold swing coefficient, and V_{off} is an empirically determined BSIM3 parameter which is also a function of threshold voltage. In these parameters, u_0, C_{OX},W/L and V_{dd0} are statically defined parameters; the DIBL factor b, subthreshold swing coefficient n and V_{off} are derived from curve fitting based on transistor-level simulations; V_{dd}, V_{th} and $v_t = kT/q$ are calculated dynamically in the simulations.

The above equation is based on two assumptions:

1. V_{gs}=0 — we only consider the leakage current when the transistor is off.
2. V_{ds}=V_{dd} — we only consider a single transistor here; the stack effect and the interaction among multiple transistors are taken into account when we model the cell using Equation 3.

Figure 1 shows the comparison of leakage current calculated by our model to the transistor-level simulation. From Figure 1a, 1b, and 1c, we can see that for the ratio W/L, supply voltage V_{dd} and temperature T, our results perfectly match the simulation results. Figure 1d shows that after threshold voltage increases to some value, the modeled leakage current does not decrease anymore. This is due to the simplicity of our model, which only considers the subthreshold leakage and DIBL effect. It is only of concern if threshold voltage is beyond the normal value.

3.1.2 Leakage per Cell

Butts and Sohi point out that their single k_{design} model is suitable only for cases where the parameters of N and P transistors are very close, and otherwise two k_{design}'s are needed. We indeed found [32] that the parameters of N and P transistors differ too much, so HotLeakage applies different k_{design} factors to the N and P transistors, k_n and k_p.

This means that for a specific cell, the leakage current is given by the following equation:

$$I_{cell\_leakage} = n_n \cdot K_n \cdot I_n + n_p \cdot K_p \cdot I_p \quad (3)$$

n_N and n_P are the number of NMOS and PMOS transistors in the cell, and I_N and I_P are the calculated leakage current of NMOS and PMOS according to Equation 2; when aspect ratio $W/L = 1$ we call them *unit leakage*. k_{design} is then a scaling factor determined for each type of cell to account for the transistor stack effect and the aspect ratios (W/L) of the different transistors. (The stack effect refers to the additional reduction in leakage when multiple series-connected transistors are off; for example, sleep transistors take advantage of this.) This means that the expression for static power analogous to Equation 1 is:

$$P_{static} = V_{dd} \cdot N_{cells} \cdot I_{cell} \quad (4)$$

k_n and k_p, the design factors of N and P transistors, can be derived by a similar method as in the single-k_{design} model. For a given cell, we divide all possible inputs into two groups: one group inputs will turn off the pull-down network composed of N transistors. The other group will turn off the pull-up network composed of P transistors. Thus the leakage currents are also divided into two groups $I_{1n}, I_{2n},\ldots, I_{kn},\ldots$ and $I_{1p}, I_{2p},\ldots, I_{kp},\ldots$ I_{kn} is the leakage current when the pull-down network is turned off, while I_{kp} is the leakage current when the pull-up network is turned off. k_n and k_p are given by the following equation:

$$k_n = (I_{1n} + I_{2n} + \cdots + I_{kn} + \cdots)/(N * n_n * I_n) \quad (5)$$

$$k_p = (I_{1p} + I_{2p} + \cdots + I_{kp} + \cdots)/(N * n_p * I_p) \quad (6)$$

N is the number of all possible combinations. For example, Figure 2 is the diagram of a two-input NAND gate. There are two

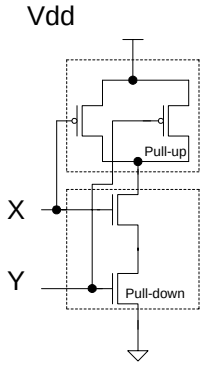


Figure 2. Two-input NAND gate.

inputs X and Y, which make four possible combinations. There are three combinations: $(X = 0, Y = 0)$,$(X = 0, Y = 1)$ and $(X = 1, Y = 0)$ which turn off the pull-down network. I_{1n}, I_{2n} and I_{3n} are the leakage currents corresponding to these three inputs. The only combination that turns off the pull-up network is $(X = 1, Y = 1)$ and I_{1p} is the corresponding leakage current. k_n and k_p are given by:

$$k_n = (I_{1n} + I_{2n} + I_{3n})/(N * n_n * I_n) \quad (7)$$

$$k_p = I_{1p}/(N * n_p * I_p) \quad (8)$$

Here N equals 4.

The double-k_{design} model has the important property that it is able to handle the differential scaling of N and P transistors that is widely used in contemporary technologies. Again, detailed plots can be found in [32]. We find that k_n and k_p are independent of threshold voltage and have a linear relationship with temperature and supply voltage. We incorporate these features into our leakage model and k_n, k_p are calculated dynamically with respect to different temperatures and supply voltages. These values are derived for different technology nodes via simulations.

3.2 Gate Leakage and GIDL Effect

In order to improve device performance, gate-oxide thickness is projected to scale aggressively for future technology nodes [27]. The result is that gate leakage through the gate oxide increases significantly due to the direct tunneling current. Our model includes gate leakage for 70nm technology, where gate leakage becomes dominant. To get an explicit equation for gate-leakage calculations is very difficult and also unnecessary for an architectural-level model. We use AIM-spice [1] as the circuit simulator, which includes BSIM4 among the supported models for gate leakage. Gate current parameters have been adjusted to target 40 nA/um gate leakage in 70nm technology at 1.2nm oxide thickness and 0.9 V supply voltage at room temperature (300K) as predicted in [27]. Gate leakage is strongly dependent on the gate oxide thickness t_{ox} and supply voltage. It is weakly dependent on the temperature. From the transistor-level simulations, we derived these factors with curve-fitting and incorporated it into our models.

GIDL effect occurs at low gate voltage and high drain voltage bias. This effect will raise the leakage current when gate voltage goes negative. It becomes worse when biasing the substrate to negative voltage for N transistors and to positive voltage for P transistors. This will limit the reverse body-biasing (RBB) technique.

3.3 Parameter Variations

Device parameter variations can be divided into two categories: inter-die (die-to-die) variation and intra-die (within-die) variation.

Inter-die variation is the difference in the value of a parameter across nominally identical dies and is typically accounted as a shift in the mean of some parameter value equally across all device or structures on any one chip. For purposes of circuit design, it is usually sufficient to lump all the contributions in the inter-die variation into a single variation component with a mean and variance.

Intra-die variation is the deviation occurring spatially within any one die. It may have a variety of sources depending on the physics of the manufacturing steps. In contrast to inter-die variation (affecting all devices on any one chip equally), intra-die variation contributes to the mismatch behavior between structures on the same chip.

Due to both inter-die and intra-die parameter variations, there is significant variation in leakage power. Thus parameter variations must be taken into account in the new leakage model. Inter-die variation can be characterized as a global mean and variance while intra-die variation is more complicated. In this version our model only includes the inter-die variation.

There are four parameters which we are interested in. They are L: length of the transistor; t_{ox}: thickness of the gate oxide; V_{dd}: supply voltage; and V_{th}: threshold voltage of the transistor. For each parameter, user can give the specific mean μ, variance σ, and the number of samples N. In the initializing phase of the simulation, N gaussian distribution samples are generated and the leakage currents are also calculated accordingly. The mean of those leakage currents is used in the following simulations in order to include the effects of the parameter variations.

3.4 How to Use the HotLeakage Software Within an Architecture Simulator

The HotLeakage simulator is a configurable module. The various parameters related to the leakage power modeling and the leakage control techniques are specified at the command line (see [32] for details). To use HotLeakage with currents based on BSIM3 models and our pre-determined values of k_{design}, it is only necessary to specify the technology parameter; e.g. 70nm. Other parameters can also be configured, but all have reasonable default values.

HotLeakage dynamically tracks leakage for each cell of interest (*e.g.*, an SRAM cell) and this information is then translated into leakage at the architecture level. The functions that calculate leakage for each structure of the micro-architecture are in the main leakage module, and these need to be called whenever any of the parameters—like temperature, supply voltage, etc.—that affect leakage is changed. These functions will recalculate the leakage currents using the HotLeakage model. HotLeakage and the accompanying simulation infrastructure currently model leakage of caches and register files; adding models for other cache-like structures is very simple.

The power-performance simulator, *e.g.* Wattch, is responsible for implementing the leakage-control technique and using the HotLeakage values accordingly. As mentioned earlier, we have implemented a generic abstraction for modeling leakage control techniques based on putting individual cache lines into standby, allowing us to study techniques like gated-V_{ss} [19], drowsy cache [11], and reverse-body-bias [23].

4 Simulation Set-Up

4.1 Processor Model

All simulations were performed with Wattch augmented by HotLeakage. Unless stated otherwise, this paper uses the baseline configuration as shown in Table 2, which resembles as much as possible the configuration of an Alpha 21264 [21]. The most important difference for this paper is that in the 21264 there is no separate BTB, because the I-cache has an integrated next-line predictor [9]. As most processors currently do use a separate BTB, our work models a separate, 2-way associative, 1 K-entry BTB that is accessed in parallel with the I-cache and direction predictor.

In the original drowsy paper, the L1 data cache used is 32 KB in size and 4-way set associative and the L1 instruction cache is 32 KB in size and direct mapped. Both caches use line size of 32 bytes and a hit latency is one. In contrast, we use 64 KB, 2-way caches with 64 B lines for both.

For Wattch and HotLeakage technology parameters we use the process parameters for a 70 nm process at V_{dd} 0.9V and 5600 MHz. It is important to note that because our Wattch model does not include state-of-the-art power-management techniques that would be expected in the 70nm generation, our estimates for dynamic energy may be pessimistic.

4.2 Benchmarks

In our comparative evaluation of various leakage control techniques, we use 11 integer benchmarks from the SPEcpu2000 [30]

Processor Core	
Instruction Window	80-RUU, 40-LSQ
Issue width	4 instructions per cycle
Functional Units	4 IntALU,1 IntMult/Div, 2 FPALU,1 FPMult/Div, 2 mem ports
Memory Hierarchy	
L1 D-cache	Size 64 KB, 2-way LRU, 64 B blocks 2-cycle latency
L1 I-cache	Size 64 KB, 2-way LRU, 64 B blocks 1-cycle latency
L2	Unified, 2 MB, 2-way LRU, 64B blocks, 11-cycle latency
Memory	100 cycles
Branch Predictor	
Branch predictor	Hybrid: 4K bimod and 4K/12-bit/GAg 4K bimod-style chooser
Branch target buffer	1 K-entry, 2-way

Table 2. Configuration of simulated processor microarchitecture. All caches are write-back.

suite. The benchmarks were compiled for the Alpha ISA and statically linked using the Compaq Alpha compiler (with *peak* settings) For each program, we skip the first two billion committed instructions to avoid unrepresentative startup behavior at the beginning of the program's execution, and then simulate 500 million committed instructions using the reference input set.

5 Results

5.1 L2 Latency

Our results roughly duplicate those in [11]. They report slightly higher leakage savings and slightly lower performance loss. The former we attribute to differences in our models, including the different choice of threshold voltage and our use of BSIM3 models. The latter we attribute to our choice of shorter decay intervals that—for our leakage model—we found to give better energy savings.

Figures 3 and 4 present the *net* cache-leakage savings and the performance loss for a system with an L2 cache latency of 5 cycles, as might be seen for a fast, on-chip L2. Note that, in order to report a measure that represents the actual "profit" in terms of energy saved, the net savings subtracts the extra dynamic energy expended due to the leakage control scheme from the total reduction in leakage that is realized by deactivating cache lines. The dynamic energy overhead is computed by comparing the total dynamic energy with and without the leakage-control scheme activated. This accounts for the contributions from (and overlap among) (a) activity in the decay counters (gated-V_{ss}), (b) extra L2 accesses (gated-V_{ss}), (c) extra tag accesses (drowsy), and (d) extra runtime.

Figures 5 and 6 then present the same results for an 8-cycle L2; Figures 8 and 9 for an 11-cycle L2; and Figures 10 and 11 for a 17-cycle L2.

These results show that for 5–8 cycle L2 caches, gated-V_{ss} is superior to drowsy cache in terms of both energy savings and performance loss. At 5 cycles, gated-V_{ss} is almost uniformly superior, while at 8 cycles, drowsy is superior for a small number of benchmarks. At 11 cycles, the picture is less clear. Gated-V_{ss} is slightly better in terms of average energy savings and slightly worse in terms of average performance loss. But looking at individual benchmarks, drowsy and gated-V_{ss} are better for about an equal number of benchmarks. Finally, at 17 cycles, drowsy cache becomes clearly superior.

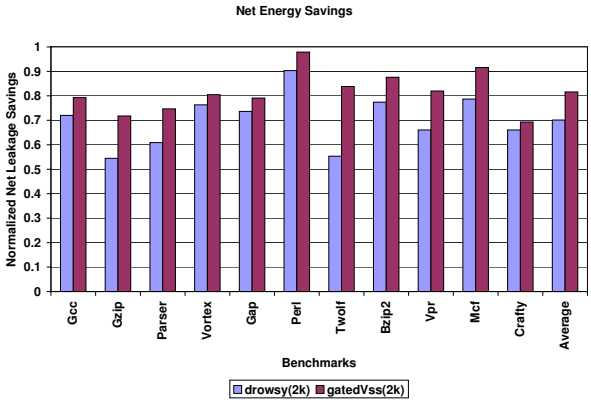


Figure 3. Net leakage savings at 110° and an L2 latency of 5 cycles.

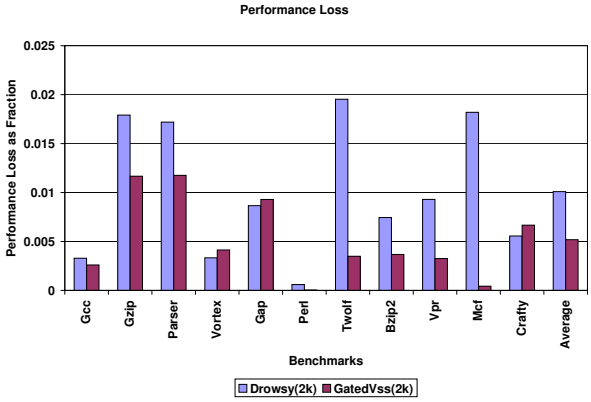


Figure 4. Performance loss at an L2 latency of 5 cycles.

Most importantly, these results show that contrary to widespread belief, non-state-preserving techniques are not inherently inferior. There are five reasons for this. First, gated-V_{ss} is able to almost entirely eliminate leakage, whereas state-preserving techniques like drowsy and RBB still exhibit a non-trivial amount of leakage. Second, a well-tuned decay interval will minimize so-called induced misses, misses that result purely from premature deactivation of a line that contains useful data. Third, induced misses are not inherently bad. Even if data remains "live", if its next use is sufficiently far in the future, it may be worthwhile to incur a modest performance loss to save energy that is otherwise expended keeping the data active. Fourth, in an aggressive out-of-order machine, modest L2 access latencies for induced misses can be tolerated. Finally, when tags are decayed, gated-V_{ss} is actually

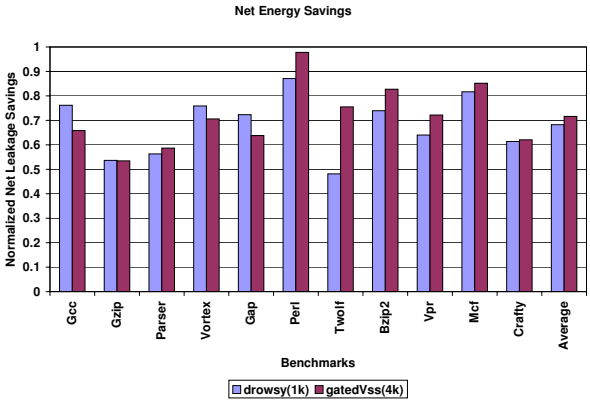


Figure 5. Net leakage savings at 110° and an L2 latency of 8 cycles.

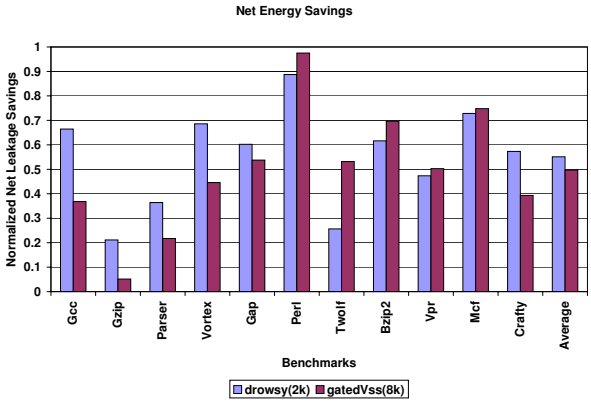


Figure 7. Net leakage savings at 85° and an L2 latency of 11 cycles.

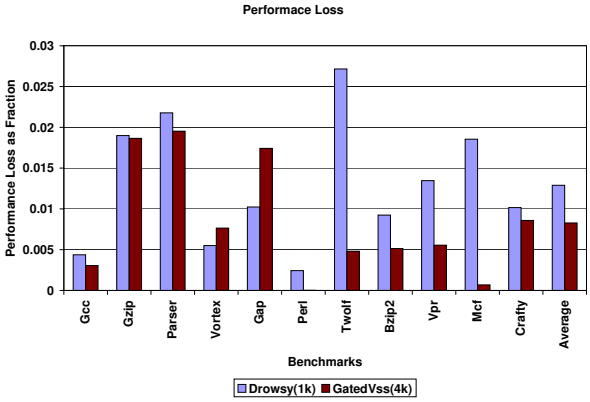


Figure 6. Performance loss at an L2 latency of 8 cycles.

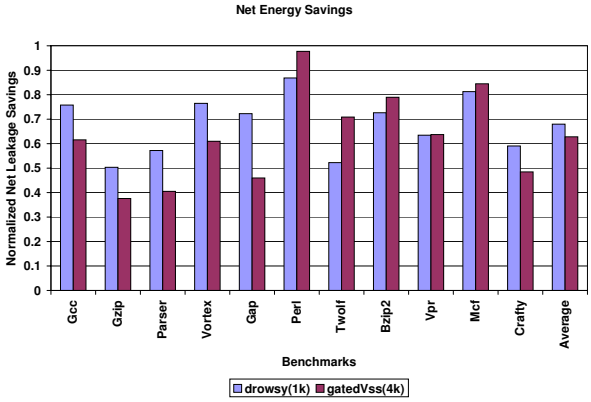


Figure 8. Net leakage savings at 110° and an L2 latency of 11 cycles.

faster on true misses when a line is in standby—which is the more common type of miss. The drowsy technique must first wake up the tags, then check them, only to find that the data is not resident and an L2 access is required. In contrast, gated-V_{ss} can immediately begin checking the tags of active ways, and ways that are in standby are guaranteed to be misses and need not be checked.

For the range of L2 access latencies that are typically observed for on-chip caches, it is therefore false to automatically assume that an L2 access is too costly. Of course, as L2 latency increases, the above factors that mitigate for gated-V_{ss} become less and less helpful. For the longest L2 latency we tested, gated-V_{ss} was no longer able to hide a significant portion of L1 miss times, and the state-preserving nature of drowsy cache becomes a major advantage.

5.2 Temperature

Figures 7 and 8 illustrate the effects of temperature for an 11-cycle L2 cache by comparing energy savings at 85°C and 110°C. Because leakage is exponentially dependent on temperature, the energy savings is much higher for both schemes.

We mentioned previously that gated-V_{ss} is able to almost entirely eliminate leakage, whereas state-preserving techniques like drowsy and RBB still exhibit a non-trivial amount of leakage. As leakage increases with temperature, this advantage for gated-V_{ss} increases too. But this advantage is offset by the fact that the

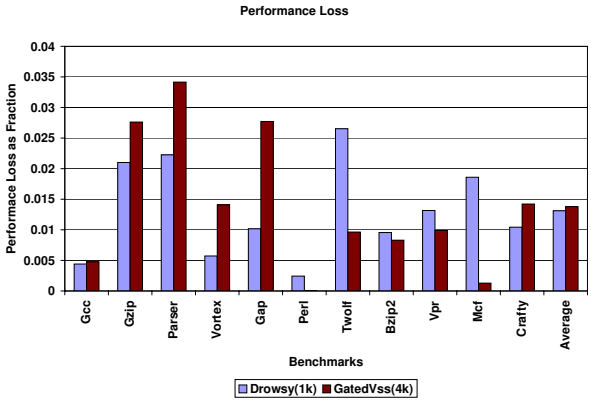


Figure 9. Performance loss at an L2 latency of 11 cycles.

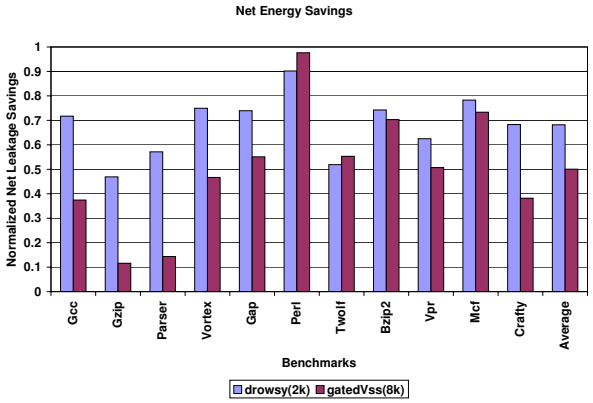


Figure 10. Net leakage savings at 110° and an L2 latency of 17 cycles.

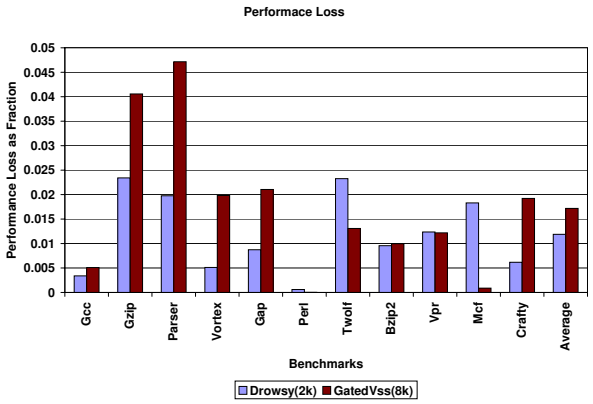


Figure 11. Performance loss at an L2 latency of 17 cycles.

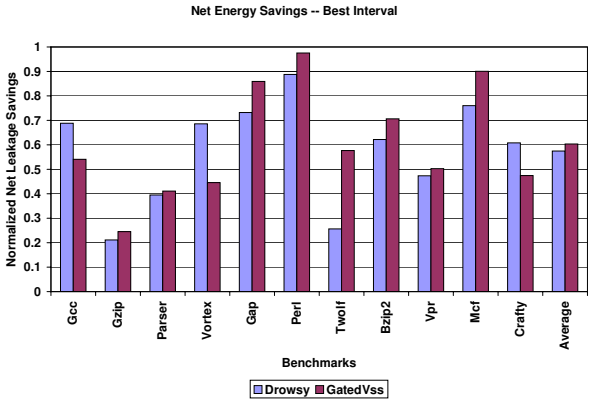


Figure 12. Net leakage savings at 85° and an L2 latency of 11 cycles for the best per-benchmark decay interval.

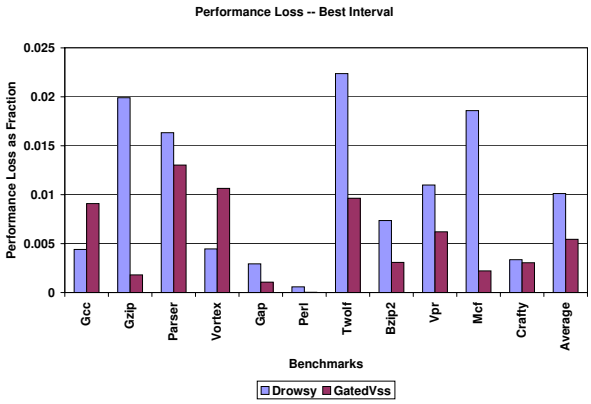


Figure 13. Performance loss at an L2 latency of 11 cycles for the best per-benchmark decay interval.

higher leakage at higher temperature makes shorter decay intervals attractive for both gated-V_{ss} and drowsy, and gated-V_{ss} is more sensitive to the smaller decay interval. The former factor benefits gated-V_{ss} for programs like *gcc* and *gzip*, but the latter factor penalizes gated-V_{ss} for *gap* and *twolf*. On average, therefore, temperature has little impact on the relative performance of gated-V_{ss} and drowsy.

5.3 Tag Decay

We have only had the opportunity to compare gated-V_{ss} when tags are also placed in standby along with the line of data that is being deactivated. If tags are not placed in standby, drowsy no longer suffers extra penalties for true misses. If one simply uses the same decay intervals but keeps the tags live for the drowsy cache, this will reduce the performance loss exhibited by drowsy but also substantially reduce the energy savings, because tags account for 5–10% of the leakage energy in caches, and this leakage energy can no longer be reclaimed. For gated-V_{ss}, on the other hand, there is no advantage to keeping the tags live unless they are used to facilitate adaptive decay intervals.

5.4 Adaptivity

Figures 12 and 13 show how much better both schemes could do if an adaptive scheme were employed to allow the cache-decay mechanism to find the best decay interval for each benchmark. For both drowsy and gated-V_{ss}, we identify the best decay interval for each benchmark, and these are the results that are plotted. The best intervals are itemized in Table 3.

Adaptivity primarily benefits gated-V_{ss}, because the best decay intervals vary so widely. This in turn is a function of data-usage patterns and available ILP that can be used to hide induced misses. Comparing Figures 12 and 13 against Figures 7 and 9 shows that using the best per-benchmark intervals improves energy savings for gated-V_{ss} by 20%, from 50% to 60%, and dramatically reduces performance loss, from about 1.4% to about 0.55%. Energy savings for drowsy cache only improve by about 4% and performance loss only improves from 1.3% to 1.0%.

It is to be expected from the analysis in [11] that adaptivity is not necessary for drowsy cache, because for reasonable intervals, it is fairly insensitive to decay interval. Gated-V_{ss} does not need

	Drowsy	Gated-V_{ss}
Gcc	1k	2k
Gzip	2k	64k
Parser	4k	16k
Vortex	2k	8k
Gap	16k	16k
Perl	4k	4k
Twolf	2k	4k
Bzip2	4k	16k
Vpr	2k	8k
Mcf	1k	2k
Crafty	4k	32k

Table 3. Best decay intervals.

adaptivity to give attractive benefits for on-chip L2 caches, but performs much better with adaptive decay intervals. It becomes clearly superior to drowsy for most benchmarks with an 11-cycle L2.

We are aware of three methods so far for providing adaptive decay intervals: using an array of bits to select from multiple possible decay intervals, proposed by Kaxiras et al. [19]; the *adaptive mode control* technique proposed by Zhou et al. [33]; and the formal feedback-control technique proposed in our prior work [31]. The latter two techniques require the tags to stay awake. Our feedback-control technique is quite simple, using the tags to identify induced misses and requiring only a small state machine to periodically update the counter containing the decay interval.

6 Conclusions and Future Work

HotLeakage provides the first publicly-available microarchitecture-level leakage-modeling software of which we are aware. Its most important features are the explicit inclusion of temperature, voltage, gate leakage, and parameter variations. HotLeakage provides default settings for 180nm through 70nm technologies (based upon BSIM3 models) for modeling cache and register files, and provides a simple interface for selecting alternate parameter values and for modeling alternative microarchitecture structures. HotLeakage also provides models for several extant cache leakage-control techniques, with an interface for adding further techniques. The HotLeakage tool, with all the supporting documents, is available at `http://lava.cs.virginia.edu/HotLeakage`

Using HotLeakage and Wattch, we have compared a state-preserving technique (drowsy cache) against a non-state-preserving technique (gated-V_{ss}). Conventional wisdom holds that the state-preserving technique must be superior, because it incurs less performance loss on access to a line that is in standby mode. In contrast, we have found that at 70nm and for the particular range of parameters we studied, the non-state-preserving technique is actually superior for a set of faster L2 cache latencies that might be seen with on-chip L2s. The main reasons for this are that gated-V_{ss} reduces leakage by a greater amount than drowsy cache, that the latency to fetch data from L2 when accessing a line in standby mode can be hidden to a significant extent by ILP, and that drowsy cache actually incurs a larger performance penalty than gated-V_{ss} for the more common case of a true (rather than an induced) miss. In addition, the effectiveness of gated-V_{ss} can be expanded by using adaptive decay intervals.

The design space for power-efficient caches is notoriously complex, and even the design space for just these two techniques is too rich to fully explore in this paper. The proper choice of leakage-control technique will depend on a variety of factors, and we hope that the comparison here illustrates some important tradeoffs to consider. The main point that we wish to convey with this work is to debunk the perception that non-state-preserving techniques are inherently inferior. Design of low-leakage caches requires non-state-preserving techniques like gated-V_{ss} to be considered as potentially the most energy-efficient and highest-performance solution.

Acknowledgments

This work was funded in part by the National Science Foundation under grant nos. CCR-0133634, CCR-0105626, and MIP-9703440, a grant from Intel MRL, and an Excellence Award from the University of Virginia Fund for Excellence in Science and Technology.

References

[1] Aim-Spice Home Page. http://www.aimspice.com.

[2] A. Alvandpour, R. Krishnamurthy, K. Soumyanath, and S. Borkar. A low-leakage dynamic multi-ported register file in 0.13um CMOS. In *Proceedings of the 2001 International Symposium on Low Power Electronics and Design*, pages 68–71, Aug. 2001.

[3] U.C. Berkeley. BSIM3 v3.1 SPICE MOS device models, 1997. http://www-device.EECS.Berkeley.EDU/~bsim3/.

[4] D. Brooks and M. Martonosi. Dynamic thermal management for high-performance microprocessors. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, pages 171–82, Jan. 2001.

[5] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 83–94, June 2000.

[6] J. A. Butts and G. S. Sohi. A static power model for architects. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 191–201, Dec. 2000.

[7] A. Buyuktosunoglu, D. H. Albonesi, P. Bose, P. W. Cook, , and S. E. Schuster. Tradeoffs in power-efficient issue queue design. In *Proceedings of the 2002 International Symposium on Low Power Electronics and Design*, Aug. 2002.

[8] A. Buyuktosunoglu, S. E. Schuster, D. Brooks, P. Bose, P. W. Cook, and D. H. Albonesi. An adaptive issue queue for reduced power at high performance. In *Workshop on Power-Aware Computer Systems*, Nov. 2000.

[9] B. Calder and D. Grunwald. Next cache line and set prediction. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 287–96, June 1995.

[10] S. Dropsho, V. Kursun, D. H. Albonesi, S. Dwarkadas, and E. G. Friedman. Managing static leakage energy in microprocessor functional units. In *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 321–32, Nov. 2002.

[11] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy caches: Simple techniques for reducing leakage power. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 147–57, May 2002.

[12] D. Folegnani and A. Gonzalez. Energy-effective issue logic. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 248–59, June. 2001.

[13] H. Hanson. Personal communication, May 2003.

[14] H. Hanson et al. Static energy reduction techniques for microprocessor caches. In *Proceedings of the 2001 International Conference on Computer Design*, pages 276–83, Sept. 2001.

[15] S. Heo, K. Barr, M. Hampton, and K. Asanović. Dynamic fine-grain leakage reduction using leakage-biased bitlines. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 137–47, May 2002.

[16] Z. Hu, P. Juang, P. Diodato, S. Kaxiras, K. Skadron, M. Martonosi, and D. W. Clark. Managing leakage for transient data: Decay and quasi-static memory cells. In *Proceedings of the 2002 International Symposium on Low Power Electronics and Design*, pages 52–55, Aug. 2002.

[17] Z. Hu, P. Juang, K. Skadron, D. Clark, and M. Martonosi. Applying decay strategies to branch predictors for leakage energy savings. In *Proceedings of the 2002 International Conference on Computer Design*, pages 442–45, Sept. 2002.

[18] W. Huang, J. Renau, S.-M. Yoo, and J. Torellas. A framework for dynamic energy efficiency and temperature management. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 202–13, Dec. 2000.

[19] S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: Exploiting generational behavior to reduce cache leakage power. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, July 2001.

[20] A. Keshavarzi, K. Roy, and C. F. Hawkins. Intrinsic leakage in low power deep submicron CMOS ICs. In *Proc. of the 1997 International Test Conference*, pages 146–55, Nov. 1997.

[21] R. E. Kessler, E. J. McLellan, and D. A. Webb. The Alpha 21264 microprocessor architecture. In *Proceedings of the 1998 International Conference on Computer Design*, pages 90–95, Oct. 1998.

[22] S. R. Nassif. Modeling and forecasting of manufacturing variations. In *Proceedings of the 2001 Asia and South Pacific Design Automation Conference*, 2001.

[23] K. Nii et al. A low power SRAM using auto-backgate-controlled MT-CMOS. In *Proceedings of the 1998 International Symposium on Low Power Electronics and Design*, pages 293–98, Aug. 1998.

[24] D. Ponomarev, G. Kucuk, and K. Ghose. Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 248–59, Dec. 2001.

[25] M. Powell, S.-H. Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar. Gated-Vdd: A circuit technique to reduce leakage in deep-submicron cache memories. In *Proceedings of the 2000 International Symposium on Low Power Electronics and Design*, pages 90–95, July 2000.

[26] K. Roy. Leakage power reduction in low-voltage CMOS designs. In *Proceedings of the International Conference on Electronics, Circuits, and Systems*, pages 167–73, 1998.

[27] SIA. *International Technology Roadmap for Semiconductors*, 2001.

[28] K. Skadron, T. Abdelzaher, and M. R. Stan. Control-theoretic techniques and thermal-RC modeling for accurate and localized dynamic thermal management. In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture*, pages 17–28, Feb. 2002.

[29] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware microarchitecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, Apr. 2003.

[30] Standard Performance Evaluation Corporation. SPEC CPU-2000 Benchmarks. http://www.specbench.org/osg/cpu2000.

[31] S. Velusamy, K. Sankaranarayanan, D. Parikh, T. Abdelzaher, and K. Skadron. Adaptive cache decay using formal feedback control. In *Proceedings of the 2002 Workshop on Memory Performance Issues*, May 2002.

[32] Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, and M. Stan. Hotleakage: A temperature-aware model of subthreshold and gate leakage for architects. Technical Report CS-2003-05, University of Virginia Department of Computer Science, Mar. 2003.

[33] H. Zhou, M. Toburen, E. Rotenberg, and T. Conte. Adaptive mode control: A static-power-efficient cache design. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2001.

Improved Modeling and Data Migration for Dynamic Non-Uniform Cache Accesses

Christopher Cowell, Csaba Andras Moritz and Wayne Burleson
[ccowell, andras, burleson]@ecs.umass.edu
University of Massachusetts Amherst

Abstract
Growing wire delay and clock rates limit the amount of cache accessible within a single cycle. Non-uniform cache access (NUCA) has been proposed as a solution to this problem in Kim et al, 2002 [1], and performance has been analyzed for various cache organizations and technology assumptions. Innovations included cache organizations which dynamically migrated data between blocks within the cache (D-NUCA) resulting in 11% improvement in SPEC2000 benchmarks over a static (S-NUCA) approach. Our work duplicates, verifies and extends the work of [1] in the following ways: 1) a commercial microprocessor, the Compaq Alpha 21364 is used for a realistic floorplan (an admitted limitation by the authors of [1]), cache sizes and wire delay estimates, 2) process technology nodes 130nm, 90nm and 65nm are used to explore the scaling of the proposed approach, and 3) new topologies and policies are developed for migrating data within the cache. Our results generally corroborate those of [1] and show that the realistic floorplan results in a 16% increased performance. Furthermore, our improved topology and policies for movement of data within the cache result in still improved performance of 43%. It should be noted that there is wide variation in the improvement of the different SPEC2000 benchmarks, thus pointing to future compiler-level approaches to D-NUCA exploitation.

Our work has been supported in part by SRC Task 766 and an SRC/IBM Masters Scholarship.

1. Introduction

Interconnect is a huge problem in high performance processors and memory hierarchies [3,5,15]. Previous work demonstrated that very large uniform cache architectures are incapable of supporting a high performance processor [17]. For each technology shrink, a smaller percentage of the chip is reachable within a clock cycle [13]. In particular, slow interconnects is the main reason for stalling a fast processor when waiting for cache accesses. Cache latency will continue to attack performance as long as cache sizes are increasing and as on-chip cache access require multiple cycles due to wire delay.

1.1 Previous Work

Prior architecture research introduced multi-ported, banked and pipelined caches to overcome the penalty of long cache accesses, but each approach has its own drawback [18]. Although multi-ported cells can satisfy more requests simultaneously, the extra logic increases the chip area and timing delay per bit and the benefits quickly diminishes when more than three or more ports is supported. Banked caches allowed cache accesses to overlap but this organization is susceptible to bank conflicts when enough addresses reference the same bank. Despite the ability to pipeline cache requests, cache latencies greater than 2 or 3 cycles have proven to negatively affect performance. More recent work shows performance improvement in the access latency as cache designs progress from uniform caches to non-uniform caches in Figure 1 [1].

The UCA cache (uniform cache architecture) is the traditional cache architecture that required the same clock cycles for all cache accesses. The ML-UCA (Multi-Level Uniform Cache Architecture) is the notion of having multiple levels of cache, where the smaller cache is a subset of the larger cache structures. The S-NUCA-1 (Static Non-Uniform Cache Architecture) is the first non-uniform architecture that is evaluated. This cache organization requires direct wiring to each of the banks where each bank is assigned a specific latency for every bank access. The second non-uniform cache architecture (S-NUCA-2) introduces network characteristics in cache architectures. The S-NUCA-2 represents a grid of networked cache banks that use shared busses to transmit data. Finally the D-NUCA (Dynamic Non-Uniform Cache Architecture) is an upgrade to the S-NUCA-2 where the most recently used data are stored in the banks closest to the processing core.

There are a variety of cache organizations to be explored but this research uses an S-NUCA2 configuration as a basis for evaluating the modified D-NUCA cache on an Alpha 21364 for a technology study (130nm, 90nm and 65nm) and a topology study for a torus, mesh and hypercube on-chip interconnection network. The rest of the paper will compare/contrast the architectural components of the D-NUCA systems in Section 2, followed by a description of the simulation environment and methodology in Section 3. The remaining two sections analyze the simulated results and present possible extensions to this work in Sections 4 and 5 respectively. Section 6 is the supplementary Appendix containing graphs and tables.

2. D-NUCA Components Comparison

The section describes the key architecture components that separate a D-NUCA from an S-NUCA2 system followed by the key difference between D-NUCA1 [1] and D-NUCA2 (this paper).

2.1 Data Mapping

Data mapping is the organization of data among the cache banks. The D-NUCA1 system in Figure 2a shows an 8-way set associative cache consisting of 32 banks. Each arrow represents a single way of the entire cache for each mapping scheme. The simple mapping scheme organizes each cache way to a numbered column. This mapping strategy is considered simple because the banks themselves are wired into vertical columns. The shared mapping scheme upgrades the simple mapping scheme by mapping data in such a way to equalize the average access delay for all sets. The four closest banks to the processing core (first rows of column 3,4,5 and 6) are composed of data that maps to each of the 8 cache ways.

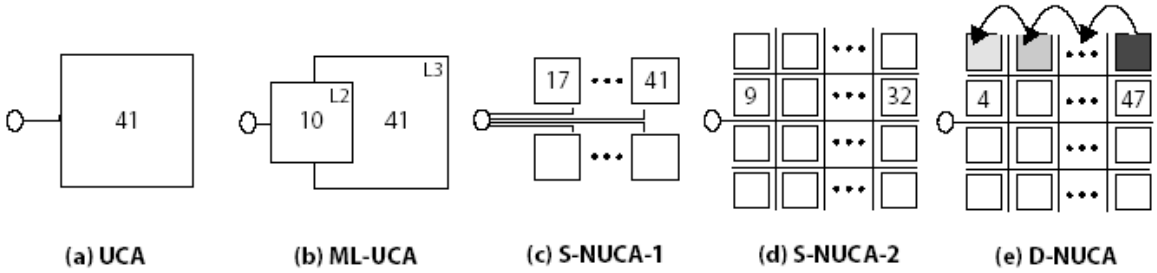


Figure 1. Cache Organizations [1]

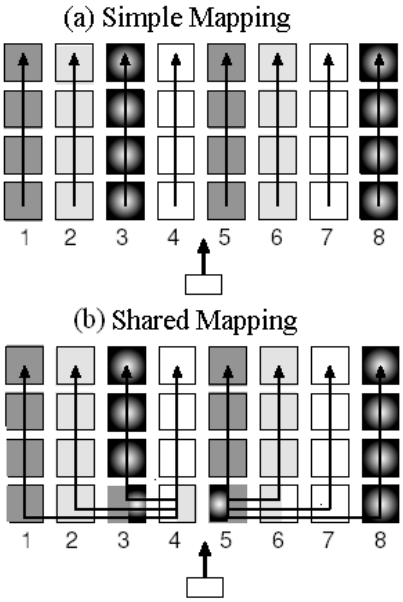


Figure 2a. Mapping Schemes [1]

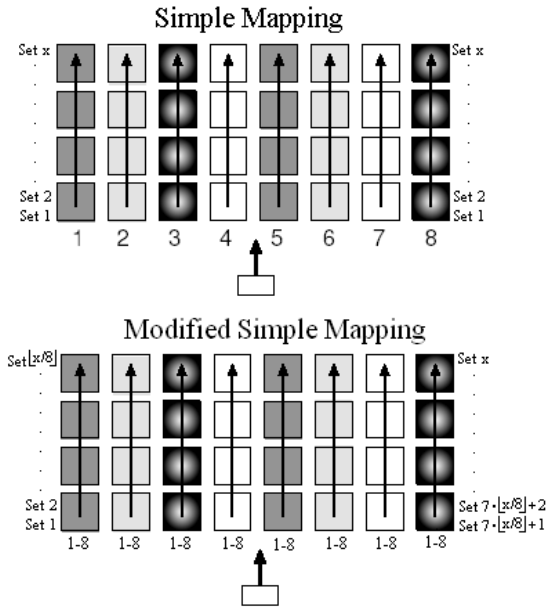


Figure 2b. Modified Simple Scheme

A simple mapping approach is preferred, because the shared mapping requires irregular wiring and data mapping to spliced banks. Because the simple mapping scheme requires little wiring overhead, there are a number of vacant wiring levels to overlay a torus, mesh or hypercube interconnection network. For this research work, data is mapped slightly different by assigning sets to individual banks in Figure 2b. This arrangement allocates all blocks within a set to a bank. Therefore a numbered set always points to a single bank. The D-NUCA2 uses a modified simple mapping scheme to allocate banks to a bank set. Therefore a modified simple approach can assign all the blocks of set 0 through set 8 to a bank versus distributing the blocks of a set amongst the banks in the original simple mapping scheme.

2.2 Bank Search

The D-NUCA1 explored two bank search policies for determining the location of a cache block. The two policies are the incremental and the multicast search. In Figure 3, the incremental search policy checks the closest bank by doing a partial tag search. If the closest bank does not generate a tag match, then a partial tag search is executed on the next closest bank and so on, until either there is a match or a cache miss. The multicast policy performs a partial tag search on all banks within a bank set in parallel. Although the multicast provides faster average access times, checking the banks simultaneously can be sensitive to contention. The multicast search policy showed the best performance and was chosen as the search policy for the D-NUCA2.

2.3 Promotion, Insertion and Eviction Policy

The original D-NUCA cache promotes data incrementally as shown in Figure 5a. As shown, a data request to a far bank triggers a promotion. When accessing a far bank, the promotion occurs when the data transmits to the processing core. If the next closest bank is full then a cache set must be demoted to the next farthest bank, essentially swapping two blocks. Their goal is to minimize the global traffic when data is swapped between two banks, by restricting data movement to neighboring banks. This current research work assumes an LRU policy that promotes data to the closest bank as the data travel to the processing core in Figure 5b and

invalidates the set in the farther bank location. In the event of a promotion and the closest bank is full, a set in the closest bank is demoted to the next farthest bank.

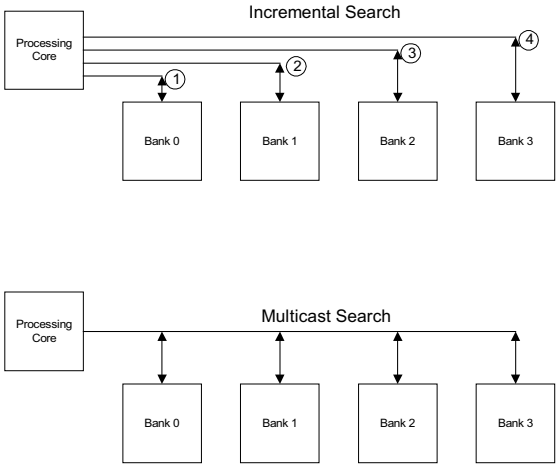


Figure 3. Bank Search Policies

Demotion continues until bank has a vacancy In order to offset the network contention that can occur during the demotion process, the research assumes dual-ported cells, a single port for reading and another for writing data. This will allow reads and writes to occur simultaneously.

The D-NUCA1 explored a number of insertion policies when retrieving new data from the lower level of cache. The best performance occurred when incoming data was inserted into cache banks that is located a moderate distance from the processing core and eviction policy that always evicts from the farthest banks. The D-NUCA2 uses an insertion policy that places new data at the head and an eviction policy that removes data from the farthest cache banks.

To summarize in Table 1, this research attempts to improve upon the D-NUCA model created at UT-Austin by introducing wire latency modeling, a more aggressive promotion policy, and the ability to interconnect banks into a variety of topologies.

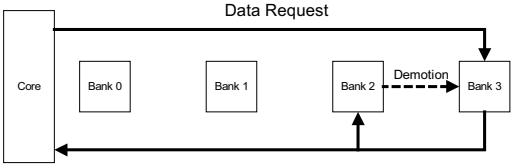


Figure 4a. Incremental Promotion

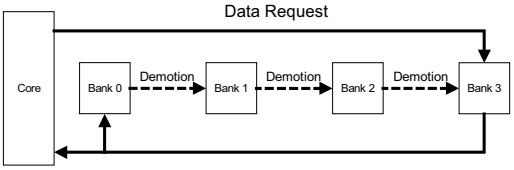


Figure 4b. Absolute Promotion

	D-NUCA1	D-NUCA2
Data Mapping	Shared Mapping	Modified Simple Mapping
Interconnection Scheme	Mesh	User Defined
Search Policy	Incremental or Multicast	Multicast
Insertion Policy	Head, Middle or Tail	Head
Promotion Policy	2-bank/1-hit	All banks/1-hit
Eviction Policy	Tail	Tail

Table 1. Simulator Model Comparison

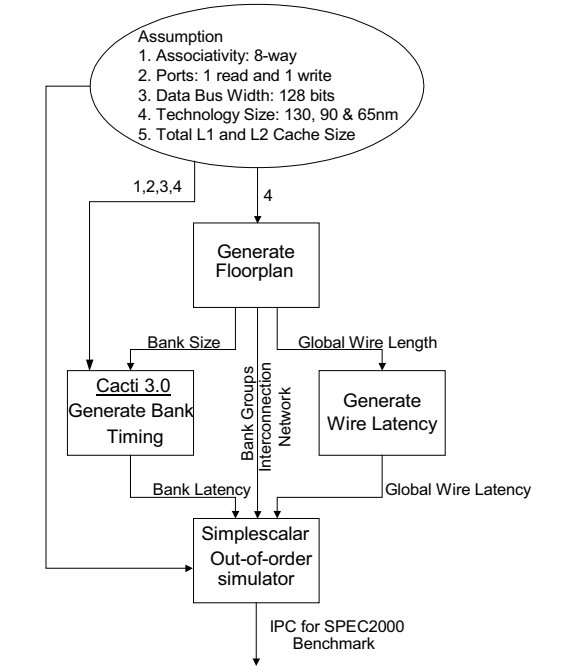


Figure 5. Simulation Flowchart

3.3 Global Wire Delay Table

The global wire delay table carries the task of converting the wire lengths extracted from a floorplan into communication delay (in cycles). The table was generated using SPICE and the ITRS 2002 (International Technology Roadmap for Semiconductor) pro-jections [2]. The ITRS projection includes the wire width, height, spacing and the dielectric permittivity. For each wire length entry, optimally placed repeaters were inserted, using the Bakoglu's method to decrease the communication delay [19]. This is common practice in the industry. The table below graphically represents only pure wire delay and does not consider the latch delay. The latch delay is considered later when converting wire delay to wire latency. Each latency conversion assumes 10% of a clock cycle to be added to the wire delay. Therefore a wire length of 0.8cm in 90nm technology would translate into 1.98 cycles, but with the inclusion of the latch delay, the total delay in cycle would rise above 2.00 cycles. Applying the ceiling function would finally bring the latency to 3 cycles, because microprocessors are not partial-cycle driven.

This access delay plus the communication delay are entered into an extended version of Simplescalar that supports non-uniform caches and the global routing delay to and from the ports of a cache bank. Simplescalar then executes a set of benchmarks using the new parameters to generate performance statistics later shown in Table 2 and 3.

3.4 Simplescalar Extended

Simplescalar is an architecture simulator that will model the Alpha 21364. A few modifications were made to Simplescalar to also model a dynamic non-uniform cache system with wire delay support. In the extended version of Simplescalar, the use is capable of specifying the quantity, size and the optimal transmission delay for each cache banks. The examples below are the necessary parameters to simulate the cache system.

cache:latency_array{
0:255:1:511:2:767:3:1023:3:1279:1:
1535:2:1791:3:2047:4:2303:1:2559:2:2815:3:3
071:4:3327:1:3583:2:3839:3:4095:4}

In the above example, the option specifies 4096 sets that are partitioned into 16 sub-banks. The first bank can hold up to 256 sets starting with set 0x000 which requires a communication latency of 1 cycles, the next bank can also hold 256 sets but starts with set 256 (or 0x100) with a communication latency of 2 cycles. Since all banks are restricted to the same size, all bank access delays are constant.

cache:bank_set{
Bank15, Bank14, Bank13, Bank12: Bank11, Bank10, Bank9, Bank8: Bank7, Bank6, Bank5, Bank4: Bank3, Bank2, Bank1, Bank0}

cache:alt_path {
Bank3, Bank7, Bank11, Bank15: Bank2, Bank6, Bank10, Bank14: Bank1, Bank5, Bank9, Bank13: Bank0, Bank4, Bank8, Bank12}

In the event that a node is busy servicing a request, depending on the inter-connection scheme, it is possible to reroute the data to the processing core. The above two parameters define the bank sets and the alternate path for rerouting around a busy node is necessary. For the configuration above, there are 4 defined search paths. When cache is read, a preliminary tag comparison determines which bank set to search and initiates a multicast. In the event that a cache hit occurs and the data collides with a busy node then an alternate route is chosen for the data to travel. The only constraint is that data can only be rerouted between neighboring (point-to-point) cache banks. Therefore in the above *alt_path* parameter, a reroute can be performed between **Bank3** & **Bank7** but not between **Bank3** & **Bank11**.

4. Simulation Results

The section compares the performance of a S-NUCA to a D-NUCA2 system. The research generates statistics for 130nm, 90nm and 65nm, and a topology study of a D-NUCA2 system that supports a torus, mesh and a hypercube.

4.1 Technology Trend

The technology trend uses a 21364 floorplan as the basis for comparing a S-NUCA to a D-NUCA. The Alpha 21364 is a model of the Alpha 21264 with large on-chip L2 caches and multiprocessor support. The results of Table 2 demonstrate that the IPC generated for each benchmark was unaffected much by communication delay and that pipelined cache accesses could easily hide the wire delay overhead. These results were somewhat expected since global delay is around a cycle for most point-to-point transmissions. The average IPC improvement was a miniscule 0.25% despite a noticeable miss rate. This implies that pipelined cache access is capable of hiding small multi-cycle delay (less than 3 cycles) within a sizeable cache structures [18].

The Alpha floorplan (90nm) in Figure 8 is a 21364 with 8MB of L2 cache. Figure 8 shows a considerable smaller processor core that is under 50% of the original core. The 90nm processor core also consumes a smaller percentage of the chip area because of the growing chip area per process generation [2]. The unused area of the chip is filled with 0.125MB cache banks. The cache bank size was reduced to make more room for supporting hardware when scaling to a 90nm process.

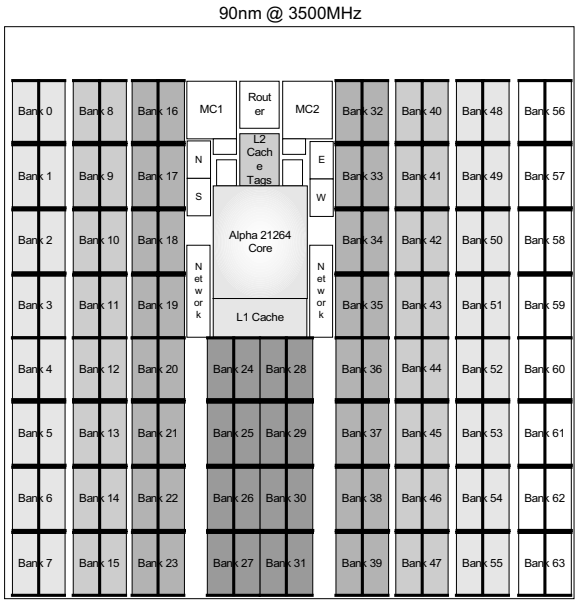


Figure 8. Alpha 21364 Floorplan 90nm

The 90nm version of the Alpha 21364 has 8MB of 64 banks each contain 0.125MB memory modules. The banks are organized in groups of 8 banks creating cubic nodes across the chip. The groups are further broken down into two subgroups of four banks. The two subgroups are restricted from exchanging data but are interconnected for routing purposes described in Section 3.

The average IPC improvement showed a 43% improvement across the benchmarks with the exception of two benchmarks. This is characteristic of excessive collision between far read accesses and data demotions from closer to farther banks. This implies that the data set is

small enough to fit inside the L2 cache banks. The low D-NUCA2 average miss rate from the 130nm to the 90nm floorplan also confirms this. /The technology study in Figure 10 correlates the throughput of an S-NUCA and D-NUCA2 for some benchmarks in SPEC2000. The study explores the IPC for 2MB, 8MB, and 16MB with a corresponding floorplan in 130nm, 90nm and 65nm. Each of the benchmarks shows a significant improvement for D-NUCA2 systems. Surprisingly for most benchmarks, the D-NUCA2 for 90nm outperforms the S-NUCA for a 65nm.

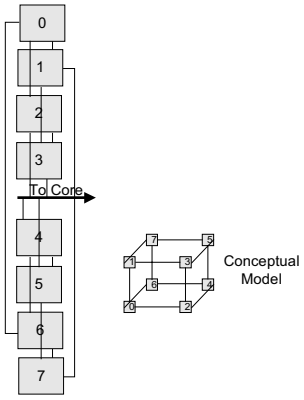


Figure 9. Cubic Interconnection Scheme

In general, D-NUCA2 shows significant improvement over a S-NUCA for large cache systems. But the D-NUCA2 shows some limitations on performance when technology migrates to 65nm. At 65nm, where the simulated cache size is 16MB, the cache system is broken down into 128 cache banks. At this stage, bank contention becomes an issue and prevents data from taking the shortest path to the processor. Research results show that this occurs frequently in 65nm technology and on occasion for *apsi* and *mgrid* in 90nm where bank contention negatively affects performance.

4.2 Topology Study

The topology study shows the performance trend for a D-NUCA system connected in a torus, mesh and hypercube network. The number of available wiring layers and the possibility of reducing the average latency motivated the idea of this topology study. As expected each of the benchmark showed an improvement as the interconnection network complexity increased. Because of the increased wiring complexity, data was less likely to stall because of a flexible network that is very capable of rerouting the data to the processor. For this reason the torus performs poorly.

Given a single node, data can only travel to another single node. In a mesh and hypercube configuration, a piece of data has the option of one or two other nodes for rerouting, respectively. The hypercube outperforms the mesh network by providing reroutes with fewer hops. This translates into a smaller average latency in Figure 11a and b.

5. Conclusion

Non-uniform cache access (NUCA) has been proposed as a solution to this problem of wire dominated cache access in Kim et al, 2002 [1]. Our works attempts to reproduce their research environment on an already existing chip floorplan as well as extend and defend their concept with architectural enhancements and a wire topology study. In general, our results corroborate those of UT-Austin and in using their best-reported configuration were able to boost performance by 43% when using a multi-cube bank interconnection scheme.

The strength of our simulation environment is the extraction of wire delay from an existing floorplan and simulated wire latency using Hspice and ITRS 2002 assumptions. The future work includes improved evaluation techniques that involve much longer simulations, studies that vary the cache bank size for very large on-chip caches. For these results, SPEC2000 was used and a simulation method that simulated the execution of 200 million instructions after fast-forwarding the 100 million instructions. Finally, because contention was an issue for very large caches, varying the cache bank size can extend superb performance improvements down to 30nm where enormous amounts of cache can fit on-chip.

6. Appendix

Benchmark	S-NUCA2	D-NUCA2	Difference %	Miss Rate %
applu	1.0833	1.2742	17.6	3.80
apsi	2.3266	1.5198	-34.7	17.2
fma3d	1.3463	2.2941	70.4	0.13
gcc	0.8118	1.4136	74.1	1.54
lucas	1.1432	1.6427	43.7	0.41
mesa	0.9707	1.9829	104.3	0.14
mgrid	2.0492	1.5003	-26.8	8.91
oammp	1.2196	1.4002	14.8	4.40
oequake	0.963	1.9928	106.9	0.17
oparser	0.985	1.7556	78.2	0.62
ovpr	1.0795	1.7822	65.1	0.36
swim	1.1829	1.7965	51.9	0.50
twolf	0.9711	1.9937	105.3	0.01
wupwise	1.4514	2.0269	39.7	0.90
		Average	**43.0**	**2.79**

Table 3. Alpha 21364 90nm

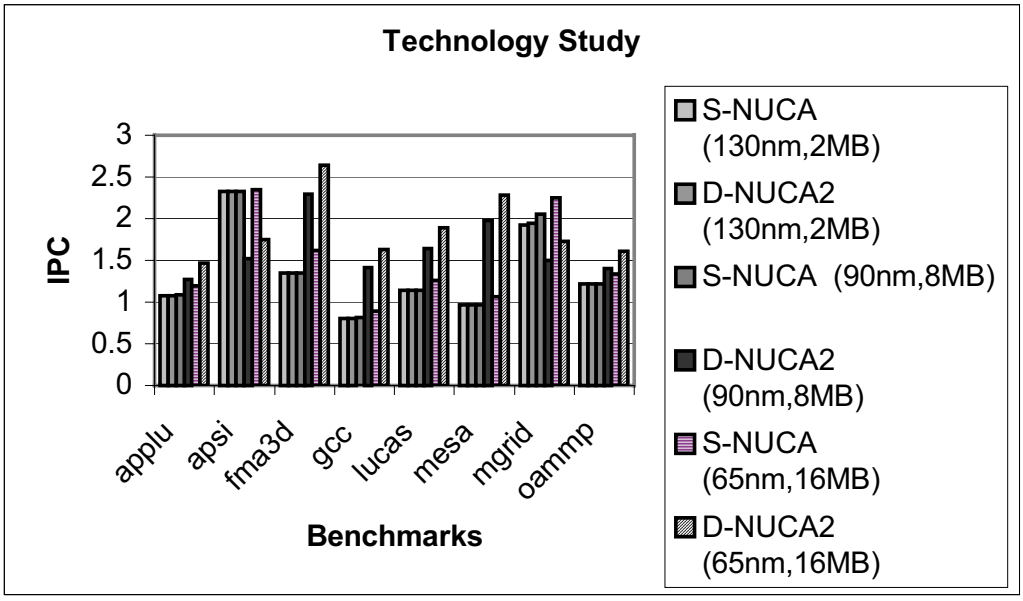


Figure 10a. Technology Trend

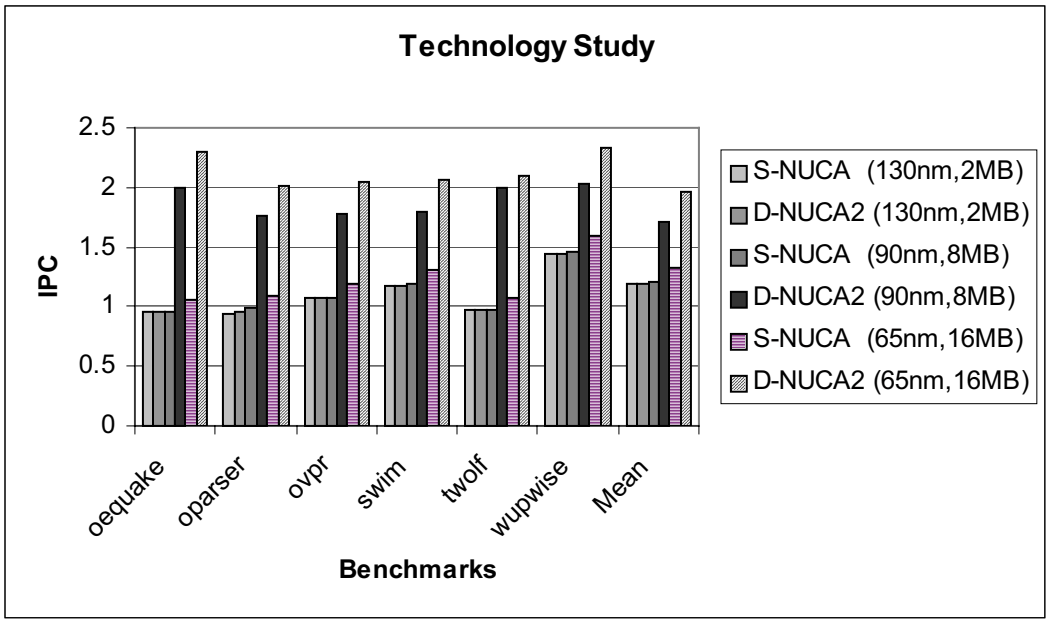


Figure 10b. Technology Trend (continued)

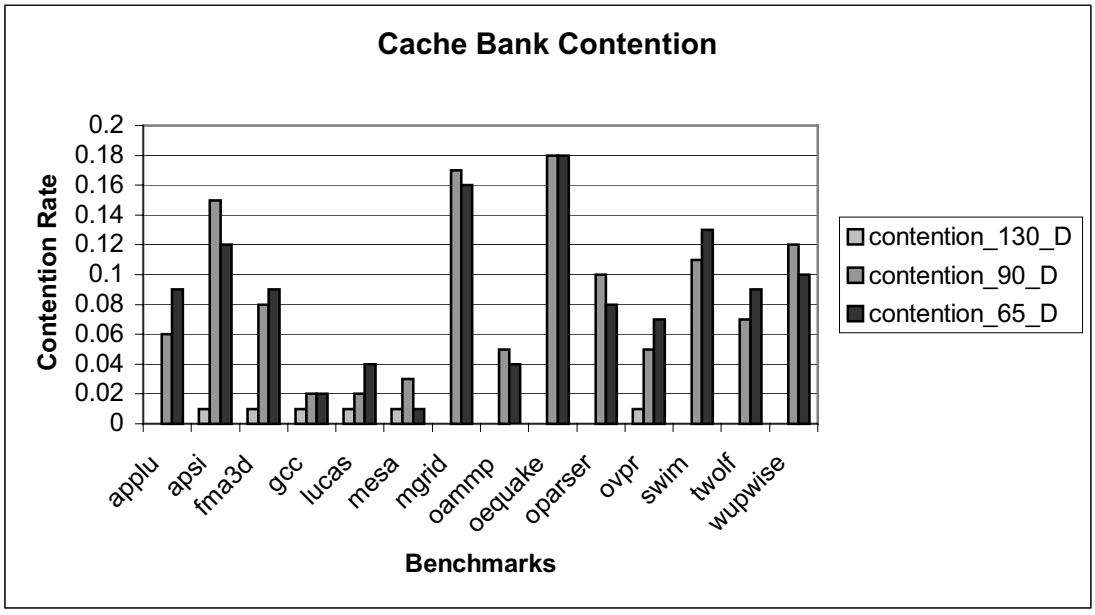


Figure 11. Cache Bank Contention for D-NUCA2

6. References

[1] C. Kim, D. Burger, S. Keckler, "An Adaptive Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches", ASPLOS, October 2002. San Jose, California.

[2] International Technology Roadmap for Semiconductors 2002 Update. December 2002. http://public.itrs.net/Files/2002Update/2002Update.pdf

[3] S. Hamilton, "Taking Moore's Law into the next Century", Computer, January 1999, pp. 43-8.

[4] D. Sylvester, K. Keutzer, "Getting to the Bottom of Deep Submicron", IEEE Computer, 1999.

[5] T. Pinkston, Y. Patt, B. Dally, B. Horst, A. Agarwal, Jose Duato, T. Basil "What will have the greatest impact in 2010: The processor, the memory or the interconnect?", IEEE MICRO 2002. http://www.usc.edu/dept/ceng/pinkston/presentations/statistics.html

[6] D. Sima, T. Fountain, P. Kacsuk, "Advanced Computer Architectures: A Design Space Approach", Addison Wesley, 1997

[7] J.L. Hennessy and D.A Patterson, Computer Archtiecture: A Quantitative Approach: 3rd Edition, Morgan Kaufmann, San Francisco, VA, 2002.

[8] D. Sima, T. Fountain, P. Kacsuk, "Advanced Computer Architectures: A Design Space Approach", Reading, MA: Addison-Wesley, 1990.

[9] A. Kleinosowski, J. Flynn, N. Meares, D. Lilja, "Adapting the SPEC benchmark suite for simulation based computer architecture research" WWC-3 pp. 73-82, 2000.

[10] S. Mukherjee, P. Bannon, S. Lang, A. Spink, D. Webb, "The Alpha 21364 Network Architecture"

[11] T. Austin, "A User's and Hacker's Guide to the SimpleScalar Architectural Research Tool Set," 1997.

[12] R. Kessler, "The Alpha Microprocessor: Out-of-Order Execution at 600MHz," Compaq Computer Corporation, August 1998.

[13] D. Matzke, "Will Physical Scalability Sabotage Performance Gains?", Computer, Sept. 1997, pp. 37-40.

[14] V. Agarwal, M. S. Hrishikesh, S.W. Keckler, and D. Burger, "Clock rate vs. IPC: The end of the road for conventional microprocessors" In Proceedings of the 27th Annual International Symposium on Computer Architecture, pages 248–259, June 2000.

[15] M. Horowitz, R. Ho, and K. Mai. The future of wires. In Seminconductor Research Corporation Workshop on Interconnects for Systems on a Chip, May 1999.

[16] P. Shivakumar and N.P. Jouppi. Cacti 3.0: An integrated cache timing, power and area model. Technical report, Compaq Computer Corporation, August 2001

[17] R. Kessler, "Analysis of Multi-Megabyte Secondary CPU Cache Memories". PhD thesis, University of Wisconsin Madison, December 1989.

[18] K. Wilson and K. Olukotun, "Designing High Bandwidth On-Chip Caches," In Proceedings of the 27th Annual International Symposium of Computer Architecture, June 1997.

[19] H. Bakoglu, "Circuits, Interconnections and Packaging for VLSI", Reading, MA: Addison-Wesley, 1990.

Virtual Memory-Induced Priority Inversion in Multi-Tasked Systems*

Gregory S. Hartman and Priya Narasimhan
Institute for Software Research International
School of Computer Science
Carnegie Mellon University
5000 Forbes Ave, Pittsburgh. PA 15213-3890
{gghartma, priya}@cs.cmu.edu

Abstract

Virtual memory (VM) sub-systems in many widely adopted desktop and server operating systems rely on approximations of the least-recently-used (LRU) heuristic to select pages for replacement. These heuristics work well when memory is abundant, but they produce counter-intuitive behavior when applications' memory demands substantially exceeds the available physical memory. This paper describes the results of preliminary experiments with a new instrumentation framework that observes Linux VM behavior in a controlled setting. Repeated experiments with a micro-benchmark consistently reveal three types of misbehavior. First, the CPU scheduler's intended priorities can be inverted for an indefinite period of time when low-priority processes push higher-priority processes out of memory. Second, the VM heuristics can perpetually assign unequal amounts of memory to simultaneously running, identical processes. Finally, processes with modest memory requirements experience execution delays during periods of memory shortage.

Keywords: instrumentation, interactive, latency, LRU, memory shortage, page replacement, priority inversion, responsiveness, unfairness, virtual memory

1 Introduction

Virtual memory [3] is a well-known source of performance problems in computing systems. Designers of real-time systems typically choose operating systems without virtual memory support [7], or lock their applications into memory to avoid the potentially unbounded delays introduced by page faults [1]. However, eliminating virtual memory support is not an option for general software development, since doing so would force programmers to predict, precisely and ahead of run-time, the amount of memory needed by their software. In particular, web servers use software where the demand for memory is extremely difficult to predict. These systems use virtual memory to provide best-effort performance, given the available physical memory. When sufficient memory is not available, these systems are subject to performance problems, including thrashing, uneven partitioning [10], and lack of performance isolation. The first author of this paper confronted the following problems while supporting a high-volume commercial web site:

1. The failure of a single process running on the web server machine could make the entire web server unresponsive. At times, the performance degradation was so severe that it was difficult to gather the data needed to isolate the cause of the failure.

2. Estimating a process' demand for memory was difficult. The operating system provided no accurate indicator of the demand and, therefore, programmers were forced to estimate their applications' memory demand using knowledge of the application. This increased the risk of failures after upgrades; software which was technically correct occasionally failed to meet the performance requirements of the web site.

3. The data that the web servers used to process requests was created through a lengthy, resource intensive conversion process. The technicians who monitored this process observed that the memory partitioning on the systems they used to convert the data was difficult to predict. To meet their schedules reliably, the technicians were forced to run no more than a single conversion at any given time on a system. In addition to forcing the purchase of expensive hardware, this technique increased the burden on system administrators,

*The research work reported in this paper has been sponsored in part by the Army Research Office under Grant No. DAAD19-01-1-0646.

made the job of the technicians more confusing, and caused congestion on the network, which had to cope with large data transfers between the systems.

The complexity of the software on the web site made it difficult to reproduce these problems reliably and to distinguish between effects caused by the application code and the operating system.

We wanted to understand the operating system's contribution to these problems, and we wanted to be certain that our results were reproducible. Therefore, we developed an instrumentation framework and an automated testing system to observe the behavior of a trivial micro-benchmark. This paper describes the design of our instrumentation framework, and the statistics that we gathered. We also describe the kind of virtual memory imbalances that we encountered, and the conclusions that we drew from our observations and results. This paper demonstrates that simple micro-benchmarks can produce virtual memory misbehavior that is similar to the misbehavior observed with real-world applications.

2 Statistics of Interest

The operating system statistics of interest to our experiments are those related to virtual memory behavior, and are listed in this section. Our instrumentation framework examines the cumulative statistics given in Table 1 once every second in order to extract the following interval statistics for each running application on the system:

- *Resident set* (denoted by `pages` on the graphs) gives the number of the application's pages that are currently in memory.
- *Zero-on-write* is an optimization that allows the operating system to defer erasing newly allocated memory. Instead, the operating system maps an erased page multiple times into the application's address space with read-only permissions. On the first write, the operating system intercepts the minor fault, allocates a new page, and maps it as a read/write page. The `zow` field on the graphs indicates the number of page faults associated with the zero-on-write optimization.
- *Percentage CPU:* (denoted by `% CPU` on the graphs) gives the number of timer interrupts that occurred in the application. The timer on Linux generates an interrupt at a frequency of 100Hz; the monitor reads the statistics once per second, so the number of timer interrupts incurred by the application, within the monitoring interval, can be directly interpreted as the % CPU load. However, execution delays in the monitor can cause this statistic to exceed 100% for an application because timer interrupts will continue to accrue during the delay.
- *Major faults:* (denoted by `majflt` on the graphs) occur when the application attempts to access a page that is not in memory. The operating system suspends the application, and initiates a read for the page from the disk.
- *Minor faults:* (denoted by `minflt` on the graphs) occur when the application attempts to access a page and the page is discovered in memory. The zero-on-write optimization can cause this to happen. Minor faults also occur when the application accesses a page on the operating system's free list before it is reallocated.

In addition, the monitor extracts the following interval statistics by querying the global cumulative statistics given in Table 2:

- *Swap-in:* counts the number of pages read from the disk for all running applications.
- *Swap-out:* counts the number of pages written to disk for all running applications.
- *Jitter:* measures the variation in the time between the monitor's samples. Jitter is normalized so that it is zero in the expected case. Our monitor process uses an interval timer to avoid accumulating the delays, so small values of positive jitter are often followed by a corresponding negative value.

3 Instrumentation Framework

3.1 Design Objectives

Our objectives in the design of the instrumentation framework included:

- *Constant monitoring:* The monitor should minimize its use of system resources so that our framework can run continuously, even on production systems without creating performance problems of its own. Constant monitoring of a production system has two advantages. First, it allows us to gather data for intermittent performance problems which may be difficult to reproduce on a development system. Second, the monitor could be extended to alert system administrators to performance problems in a timely manner, ultimately reducing the time needed to correct the problem.
- *Reflective monitoring:* To be comprehensive, we required that our monitor process keep statistics for itself. This allows us to keep track of the monitor's own

resource consumption. The data that the monitor reads is updated by the operating system independently of the monitor process. If the monitor is blocked (for reasons such as page faults), there will be artifacts in the reported data. By tracking the behavior of the monitor, we can locate these artifacts, and exclude them from our analysis.

3.2 Assumptions

We made the following assumptions while designing our framework and while running our tests:

- The system always enters the same state after a reboot. Therefore, identical tests started at the same time after a reboot will produce identical results. We test this assumption by running the same test many times, and by comparing the results. Where the results differ, we document the differences.
- The monitor process does not affect the behavior of the system to the extent that our findings cannot be applied to systems that are not running the monitor. We have been able to reproduce the behavior described in this document even without the use of the monitor.
- The entries in the `/proc` file-system [8] that we use for gathering the kernel statistics accurately reflect the state of the system, as documented in the Linux man pages. We have examined the Linux kernel source code for the statistics of interest to us, and verified that the `/proc` implementation corresponds to its documentation.
- Linux has not been designed to favor processes based on their starting time. Therefore, unequal resource allocations to multiple, identical simultaneously executing applications constitutes undesirable behavior.

3.3 Implementation Details

Our framework runs over a Red Hat Linux 8.0 installation. We replaced the kernel with an enhanced stock (non-Red Hat) 2.4.20 kernel that provides additional statistics that allow us to observe the behavior of the page-reclamation algorithms. The test system has a Pentium 4 processor running at 2GHz with 512MB RDRAM memory and a 40GB IDE hard disk. Our instrumentation framework, consisting of a monitor, a test script and some kernel-level instrumentation, attempts to pinpoint virtual memory imbalances through a micro-benchmark that we designed.

Micro-benchmark. Our micro-benchmark is a simple memory scanner. It allocates a single 384MB buffer using the `malloc` routine in the C runtime library, and then writes sequentially to the entire buffer one byte at a time. When it gets to the end of the buffer, it returns to the beginning of the buffer, and continues to write sequentially. Instances of the micro-benchmark are called scanners in the rest of this paper. One scanner will clearly fit into the physical memory (512 MB) of our system; additional scanners will undoubtedly cause paging. This scanner was originally intended to validate memory allocation in order to test the statistics gathered by our monitor. Although the our current micro-benchmark provides valuable insights, we intend to apply our framework to more realistic workloads and applications in the future. We implemented the scanner on Linux, and have additionally ported it to FreeBSD and Windows XP.

Monitor. The monitor is a process that queries the `/proc` file-system once every second to extract the kernel statistics for every running process, and writes the values to a log file. The monitor gathers global statistics, such as the number of free pages, as well as process-specific statistics, such as the number of resident pages. The monitor currently uses the Linux `/proc` file-system to translate internal kernel data structures into a text representation. The per-process and global statistics of interest to us are listed in Tables 1 and 2, respectively. We note that the monitor does not gather the information in an atomic operation, so discrepancies can appear in the data. The monitor records timestamps from the CPU clock register to allow us to detect jitter between the samples. It also records, in the log, the first appearance of a process, and also a process's exit from the system.

The monitor also launches scanners. The first scanner is launched 12s after the monitor starts. After the the monitor launches a scanner, it waits for a fixed time interval before launching an additional scanner. The delay between launches and the number of scanners to launch are randomly selected at the beginning of each test-run. After running for one hour, the monitor terminates the test-run by rebooting the system.

Test Script. This script runs during the boot sequence on Linux. It halts the boot process, and launches the monitor before the networking sub-system starts, thereby minimizing the number of processes on the system, and also eliminating any network traffic that might perturb our experiments. The monitor and this script constitute a fully automated test system, greatly reducing the variability of the timing of the tests. The other processes running in the system at the time of the test include: `init`, `keventd`, `kapmd`, `ksoftirqd_CPU0`, `kswapd`, `bdflush`, `kupdated`, `mdrecoveryd`, `kjournald`, `rc`, `minilogd`, `initlog`, and `bash`. The `bash` process runs on the console, allowing the test to be terminated so that results can be collected from the system.

Label Name	Description
comm	Program name
ppid	UNIX process identifier of the parent process
minflt	Number of the page faults that didn't result in a disk I/O
majflt	Number of page faults that resulted in disk I/O
utime	Time spent in user-level code for this process
stime	Time spent in the operating system for this process
priority	Process priority for the Linux scheduler
rss	Number of pages in the page table of the process
zerofilled	Number of copy-on-write operations of the process
reclaimed	Number of pages reclaimed from this process
reclaimscans	Number of times that the kernel attempted to reclaim pages from the process

Table 1. Per-process statistics of interest. The monitor process logs a copy of these kernel counters for each task on the system. The `zerofilled`, `reclaimed`, and `reclaimscans` counters are our enhancements to the kernel to understand the behavior of the Linux page-swapping system.

Label Name	Description
memtotal	Total amount of memory in system, as reported from `/proc/meminfo`. (in kB).
memfree	Free memory, as reported from `/proc/meminfo`. (in kB).
memshared	Shared memory, as reported from `/proc/meminfo`. (in kB).
swapin	Number of pages read from swap files.
swapout	Number of pages written to swap files.
cycles	Number of processor cycles (w.r.t. our 2GHz testbed) since the last read.
readtime	Number of cycles spent reading the data from the `/proc` file-system.
fairness	Value of the fairness heuristic of any memory scanners; -1 if no scanners, 0 if completely fair, 1 if completely unfair.

Table 2. Global statistics of interest. This is a partial list of the counters that track the behavior of all the processes on the system. The statistics in bold are generated by the monitor.

Kernel Instrumentation We enhanced the Linux kernel with counters to track the behavior of the virtual memory sub-system as it allocates and reclaims pages. These additional counters are `zerofilled`, `reclaimed`, `reclaimscans`, and are described in more detail in Table 1. Before we added these statistics, the kernel did not provide any indication that memory had been reclaimed from an application.

4 Empirical Results

Using our framework, we have observed three behaviors that make virtual memory performance difficult to predict:

1. *Priority inversion* - the virtual memory sub-system does not respect the CPU scheduler's process priorities. Specifically, a low-priority process can steal pages allocated to a higher-priority process. In certain circumstances, the low-priority process might retain these pages indefinitely, suspending the higher-priority process when the latter encounters page faults. As a result, the lower-priority process obtains virtually exclusive CPU access, leading to priority inversion.

2. *Virtual memory imbalances* - identical applications obtain significantly different allocations of the available physical memory when they are run together. These imbalances exist in the operating systems that we have tested (Linux, FreeBSD, and Windows XP), and appear to be an artifact of the victim-selection algorithm. Our detection of virtual memory imbalance hinges on (i) our knowledge of our micro-benchmark's memory-related behavior, (ii) our use of three *identical,* simultaneously executing instances of our micro-benchmark, and (iii) our ability to observe unfairness in physical memory allocations across these instances.

3. *Execution delays* - processes may exhibit pauses on the order of one second when memory is scarce. This is true even of processes that run periodically, and that have small, predictable page-reference strings. The monitor process that we use to examine the Linux kernel counters for the purpose of tracking resource allocations on Linux was originally susceptible to this problem. We were able to work around this problem by locking the monitor into memory. While this approach does work for other applications, it may be infeasible for applications which use a large amount of memory, or those that process a mixture of high-priority and low-priority requests.

These problems stem from the page-replacement algorithm's ability to affect which applications experience page faults on multi-tasking systems. This, in turn, affects task scheduling, which ultimately affects the global order of page references in the system. When the page-replacement algorithm is driven from this global order, as is the case in the least-recently-used (LRU) heuristic [2] used by the virtual memory sub-systems in Linux [5, 6] and FreeBSD, a feedback loop can occur. While these effects have been known for some time [10], many operating system texts (for example [9]) present an oversimplified view of the virtual memory sub-system which does not address this problem. Unfortunately, the LRU heuristic penalizes applications with conservative memory and CPU usage in favor of applications that allocate memory and use the CPU liberally.

Our instrumentation framework also indicates that the existing kernel statistics do not provide enough information to quantify either an application's memory demands or the amount of free memory on the system. We demonstrate that the statistics in the Linux kernel can either over-estimate or under-estimate an application's demand for memory, depending on the available memory in the system at the time. To make matters worse, the accuracy of the statistics seems to improve only when memory is reclaimed from applications, which often results in execution delays.

We work around this limitation in our current research by observing multiple instances of a micro-benchmark which has a static, pre-specified demand for memory. Any difference in the runtime allocation of memory across the identical instances, therefore, serves as an indicator of imbalance in memory allocation. Real-world applications are not so simple; their memory demands may vary based on the stage of processing (for batch applications) and the input (for interactive applications). In addition, few systems are exclusively dedicated to running multiple copies of the same application. Therefore, as a part of our next research steps, we intend to find a better way of characterizing an application's demand for memory. Our conjecture is that we can enhance the kernel, with minimal impact on application performance, to observe the working-set [4] of applications, and that this working-set information will accurately reflect the memory demand of applications.

4.1 Priority Inversion and VM Imbalance

We describe a single test-run involving three scanners, each of which is started at different times (12s, 127s, and 242s, respectively). This test is representative of 81% of the test-runs using three scanners (*i.e.*, 36 out of 44 test-runs) that we have analyzed. The remaining 19% of the test-runs (called non-conforming tests) do not exhibit the unbalanced memory behavior; instead, the three scanners end up sharing the memory equally. We believe that the non-conforming tests are different because the system does not reach a stable state between the launches of the micro-benchmark; all of the non-conforming tests have a delay of less than 30s between micro-benchmark launches (as compared to the 115s inter-scanner interval described below).

The variation of the kernel statistics with the launch of the three scanners is shown in the graph in Figure 1.

- **Initial system behavior**: The first part of the graph shows the behavior of the system at the beginning of the test, when the system has just rebooted. The testing process pauses for 12s to ensure that the boot has completed. During this time, a large number of pages are free.

- **First scanner starts:** After these 12s have elapsed, the testing process launches the first scanner. A large number of minor faults occur between 12s and 14s; all of these seem to be caused by the zero-on-write optimization. Note that the process causes 75 major faults when it starts. Since there is plenty of memory available, the minor and major faults do not continue as the process runs. In addition, there is no swapping activity. The CPU load quickly approaches 100%, because the scanner never sleeps, and never has to wait for memory. The system reaches a stable state at 15s, and remains in this state until the second scanner starts.

- **Second scanner starts:** The monitor launches the second scanner at 127s. This scanner also causes 75 major faults when it starts; it also causes a large number of minor faults, most of which are associated with the zero-on-write optimization. Unlike at the launch of the first scanner, memory is now constrained. The second scanner's demand for new pages causes pages to be swapped out for the first time during the tests. In addition, the minor faults are spread over a 10s period instead of being clustered in a 3s window as they when the first scanner started. At 136s, the system begins to page in some of the memory that it swapped out. As a result, the CPU utilization falls below 5% once we are 144 seconds into the test-run.

- **Continued instability:** The system does not reach a stable point after the second scanner runs. In fact, the first scanner gains control of the memory and the CPU at 173s, only to lose it again at 203s! This event, however, is not present in all of the tests-runs using three scanners.

- **Third scanner starts:** The third scanner begins to run at 242s. Since there are few free pages at this time, its zero-on-write faults extend until 253s. There is a corresponding increase in swap-out activity as the operat-

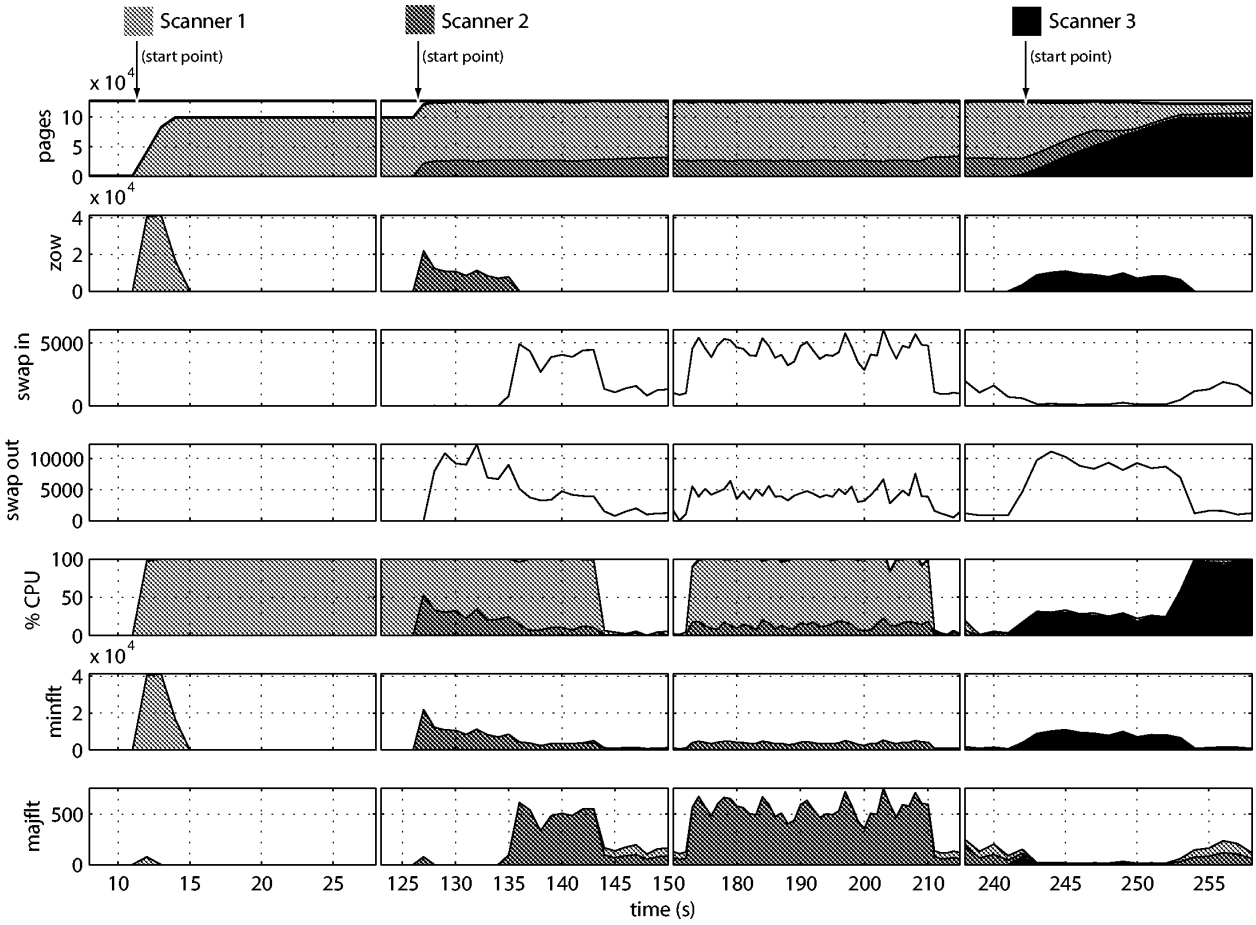


Figure 1. VM imbalance in a typical three scanner test. Column 1 shows the first scanner's launch; column 2 shows the second scanner's launch; column 3 shows a brief period of dominance for the first scanner; column 4 shows the third scanner's launch. The third scanner quickly pushes the others from memory.

ing system writes pages to disk. The new scanner consumes 20-30% of the CPU during this period of time. The third scanner's share of the memory gradually increases, until its all of its pages are in memory at 254s. At this point, the third scanner consumes nearly 100% of the CPU. While this increase occurs, there are very few swap-ins, probably indicating that the disk is saturated with the swap-out requests. The zero-on-write optimization gives the third scanner an advantage over the existing scanners; it can claim any page on the free list, zero it, and use it immediately. The first two scanners, on the other hand, must wait for data to be loaded from the disk before they can use new pages.

Once the system has reached this state, the third scanner retains its memory indefinitely, as seen in Figure 2. The third scanner can almost always run, since most of its pages are in memory. Therefore, its pages are not likely to be selected by the LRU heuristic. On the other hand, the first two scanners are almost always blocked as they wait for pages from the disk; many of their pages will, thus, be on the LRU list. This is because the LRU heuristic uses a global view of time rather than the execution times [4] of the individual scanners to age their respective pages. The problem may be addressed by employing the working-set heuristic to age a process's pages based on the process's execution time rather than on a global time-base.

The memory imbalance is so pronounced that it defeats the purpose of priorities in the Linux scheduler. This can be seen from the data in Table 3. This data indicates that at all of the sampling points beyond 260s (after the imbalance occurs and persists), the third scanner has a priority lower those of the first two scanners. This is expected behavior, because the third scanner is using nearly all of the

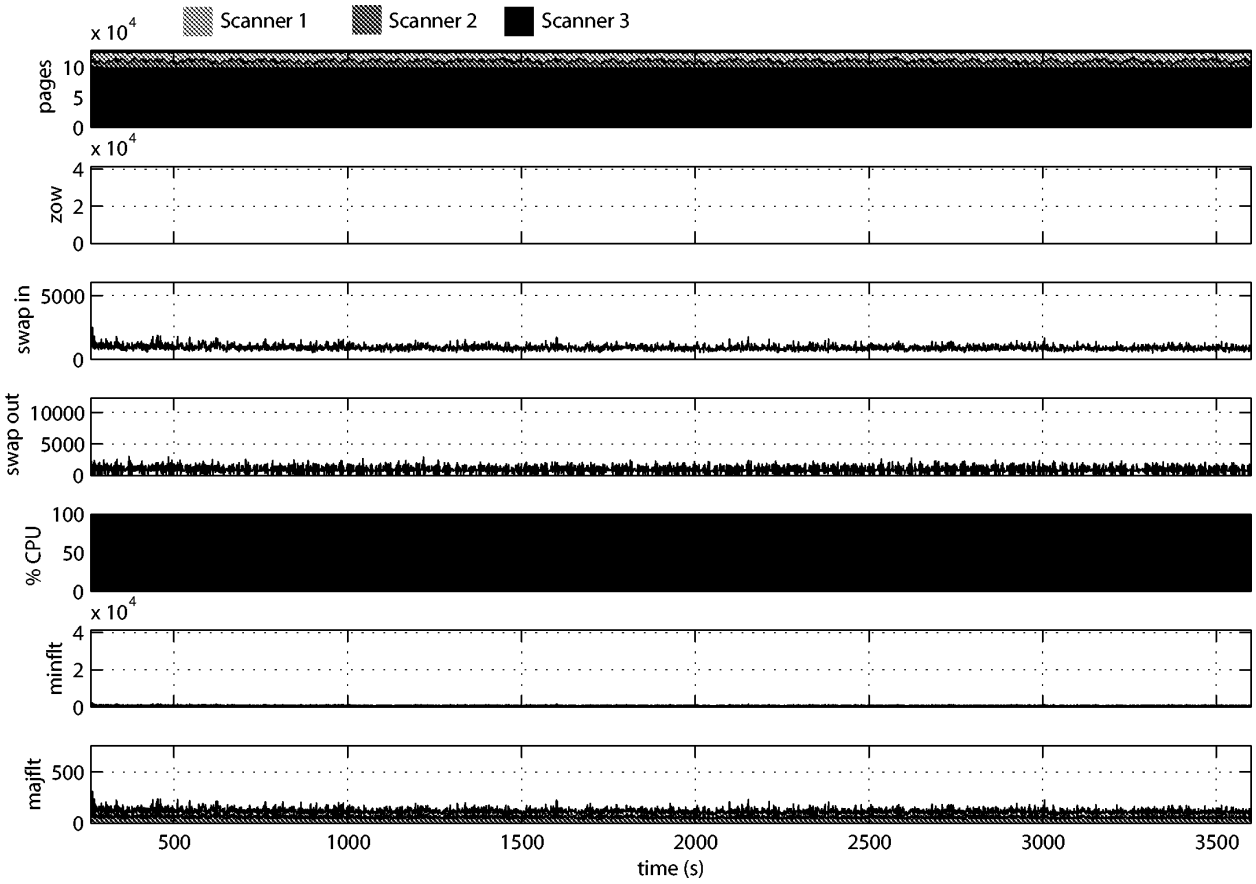


Figure 2. VM imbalance persists indefinitely.

	Min % CPU	Average % CPU	Max % CPU	Min Priority	Max Priority
1	0	1.6	8	9	12
2	0	1.6	8	9	11
3	84	96.1	107	14	20

Table 3. Priorities and CPU usages of the three scanners between 260s and 3600s of a representative test-run. On Linux, higher numeric priority values indicate lower-priority processes. The value of 107 in the Max % CPU column for scanner 3 is caused by a .07s irregularity in the scanner's sampling interval at some point during the test-run. Priority inversion occurs because the third scanner consistently runs at a lower priority than scanners 1 and 2, but gets a much larger share of the CPU.

CPU, and the default scheduler in Linux lowers the priority of such CPU-intensive processes. However, the behavior of the virtual memory system blocks the first two scanners. Therefore, they only consume only 1.6% of the CPU in spite of their status as high-priority processes, while the third scanner obtains 96.1% of the CPU. This is clearly a case of priority inversion.

This behavior does not emerge in all of the tests. In 19% of the test-runs (all of which had a delay between scanner launches of 30s or less), the memory between the three scanners was ultimately balanced. In a small number of tests, the memory spontaneously re-balances after being unbalanced for 20-40 minutes. We have not been able to reproduce this behavior consistently, and it seems to happen in approximately only 5% of the tests when the number of scanners and the delay between scanner launches are fixed. In addition, we can find nothing in our logged statistics to indicate the cause of this relatively infrequent stable behavior.

We ran some initial tests to verify that this behavior was not specific to the virtual memory implementation in the Linux kernel alone. Therefore, we ported the scanner to FreeBSD and Windows XP, and conducted our experiments. We were not able to port the monitor[1], so we ran the scanners manually, and observed the system behavior instead through the `top` utility (in the case of FreeBSD) and the Task Manager (in the case of Windows XP). The FreeBSD system exhibited the same instability that we observed on Linux. The observed behavior on Windows was somewhat different; when the second scanner starts, it is unable to recover memory from the first scanner. When the third scanner starts, the operating system drastically reduces the resident sizes of all of the scanners. This behavior may be desirable, since the memory does not reduce the faulting rate of the scanners. However, the GUI of Windows XP still exhibits large execution delays when three scanners are running. In our future work, we plan to quantify these delays and to log them in the monitor.

[1]The porting of the monitor was hindered because these operating systems do not provide the `/proc` interface for ready access of the kernel statistics.

4.2 Execution Delays

In our early experiments, we occasionally observed pauses of approximately 1.7s in the monitor output. These pauses seemed to occur just after the second scanner started. When we examined the logged data for the monitor process, we discovered that the monitor was experiencing major faults near these delays. This was somewhat surprising, because the monitor ran once every second, and "touched" the same addresses each time that it ran. On examining the data in the logs further, we concluded that the monitor needs to maintain 76 pages (304kB) of its pages in memory in order to avoid these major faults.

However, these pauses made the log files from the monitor difficult to interpret. Therefore, we decided to lock the monitor into memory, making its pages ineligible for replacement. This appears to confine the jitter to approximately 1/50th of a second, allowing us collect data even while the system is experiencing severe swapping. Figure 3 shows data from an early version of the monitor which experienced one of these pauses. In this test-run, the second scanner starts at 36s, causing the system to run low on memory. As a result, there are gaps in the data at 38s and 40s, (shown as blank spaces on the graph). At 40s, the monitor outputs a jitter value of 1.7s. This is consistent with the monitor missing two samples. The major fault data (graphed as `majflt`) shows that the monitor detected a total of 34 major faults at 39s and 41s. In this particular test-run, the monitor experiences one additional major fault. However, this fault does not seem to cause appreciable jitter in the output.

5 Predicting System Performance

The instrumentation of the virtual memory system does not allow us to make predictions about the system performance. To make reliable predictions, we would need to understand the memory demand of the applications. An application's memory demand is not equivalent to its resident size, as reported by the operating system. The working set[2] for an application may be much larger than what is indicated by the resident set in cases where the system is low on mem-

[2]"The working set is usually defined as a collection of recently referenced segments (or pages) of a program's virtual address space."[4]

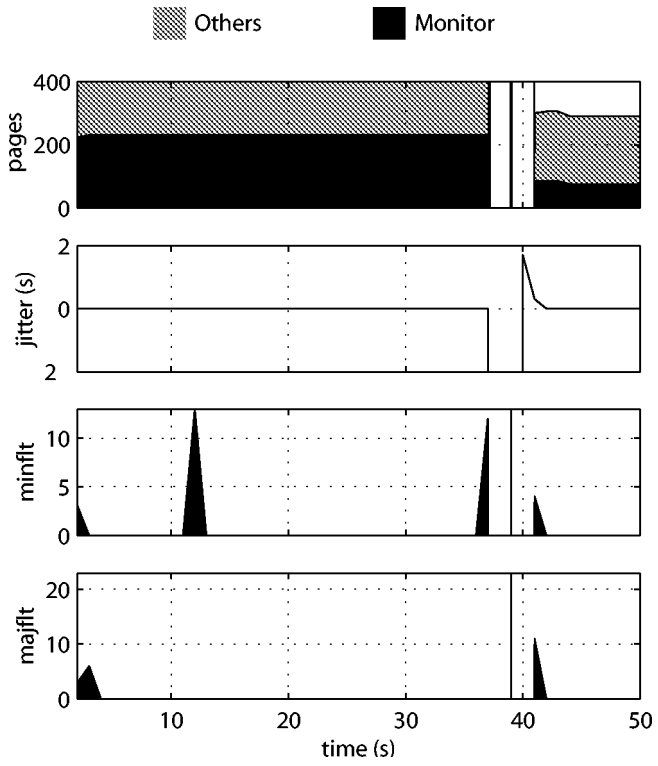


Figure 3. Jitter in the monitor process. The maximum of the y-axis on the *pages* graph has been lowered, and the non-monitor faults have been removed from the *majflt* and *minflt* graphs to make the monitor's statistics more apparent. The gaps at 38s and 40s are the result of missing log samples. Some data is available at 39s, and is represented as a thin vertical line. The 1.7s delay at sample 40 accounts for the two lost samples.

ory. This can be clearly seen in the fourth column on Figure 1. Here, the resident set accurately reflects the memory demand for the third scanner. However, the first two scanners have the same working set, but a much smaller resident set. Because these scanners generate a large number of major faults, we can conclude that the resident size is smaller than the current memory demand of the scanners. However, there do not exist kernel statistics that will tell us the difference between the current resident set and the true memory demand of the scanner. We believe that working sets represent one potential solution in order to provide a much more accurate estimate of the memory demand of applications.

On systems where the memory is not scarce, the resident set may overestimate the memory demand of an application. This can be seen on Figure 3. Here, the monitor's resident size starts out at 231 pages (see the data between 1s and 38s). However, when the number of free pages drops, the operating system begins to recover some of the pages. Once this recovery has finished, the resident size drops to 85 pages (between 41 and 43s). This is still larger than the actual memory demand since the resident size drops to 76 pages at 44s. This approach to quantifying an application's memory demand has several disadvantages:

- There is no way to know when the resident size accurately reflects the memory demand.
- The application often encounters major faults during the page reclamation, degrading its performance.
- The number of pages marked as free on the system is artificially low. This makes it difficult to predict the behavior of the system when new applications are run.

6 Future Work

We plan to continue our investigation into the relationship between the virtual memory subsystem and response times. The questions that we would like to address in future research include: What set of benchmarks reproduce the problems that realistic applications encounter on memory constrained systems? Does the size of the working-set accurately reflect an application's memory demand? Can we use this information to predict system behavior when applications are added to systems? Can we detect execution delays in unmodified applications without consuming a large portion of the system's resources? If so, how often do these delays occur, and what are causes of the delays?

7 Conclusion

In this paper, we have demonstrated that (i) it is relatively straightforward to reproduce virtual memory misbehaviors, such as priority inversion, (ii) these VM misbehaviors persist for a long period of time, (iii) identical applications can obtain radically different shares of the physical memory, depending on the state of the system when they are started, (iv) execution delays afflict all applications on the system, even those that are written to make careful use of memory, and (v) existing kernel statistics do not provide a clear picture of applications' memory demand and of the system's available memory.

Virtual memory is used on a wide array of systems, ranging from desktops to large servers. Given the information above, we conclude that applications on these systems are vulnerable to VM-induced performance failures. These failures could be triggered by unreasonable requests, buggy software, or malicious programs. In addition, these systems are difficult to configure—while a system's statistics might

indicate that it has insufficient memory, these statistics cannot be used to quantify the amount of additional memory needed to achieve acceptable application performance. Our future work will examine the behavior of virtual memory (and of other resources) in order to provide applications with a stable, predictable platform.

8 Acknowledgments

We would like to thank the anonymous reviewers for their comments that helped us to improve this paper. We also thank Tim Halloran and Jay Wylie for helping us to clarify our ideas in the final version of the paper.

References

[1] *Digital UNIX Guide To Realtime Programming*. Digital Equipment Corporation, March 1996.

[2] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, July 1966.

[3] P. J. Denning. Virtual memory. *ACM Computing Surveys (CSUR)*, 2(3):153–189, 1970.

[4] P. J. Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, SE-6(1):64–84, 1980.

[5] M. Gorman. Code commentary on the Linux virtual memory manager. Technical report, University of Limerick, 2003.

[6] M. Gorman. Understanding the Linux virtual memory manager. Technical report, University of Limerick, 2003.

[7] H. Kopetz. *Real-Time Systems*. Kluwer Academic Publishers, 1997.

[8] A. Rubini and J. Corbet. *Linux Device Drivers*, pages 103–108. O'Reilly and Associates, Inc., 2nd edition, 2001.

[9] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. John Wiley & Sons, Inc., 6th edition, 2003.

[10] A. J. Smith. Multiprogramming and memory contention. *Software–Practice and Experience*, 10(7):531–552, 1980.

Evaluating the Relationship Between the Usefulness and Accuracy of Profiles*

Geoff Langdale [1]
[1]Carnegie Mellon University
Pittsburgh, PA 15213
geoffl@cs.cmu.edu

Thomas Gross[1,2]
[2] ETH Zürich
8092 Zürich, Switzerland
trg@inf.ethz.ch

Abstract

The relationship between how accurately a profile predicts future program behavior and how useful it is for profile directed optimization is not straightforward. We gathered extensive data on the results of profile-driven optimization using two different optimization systems (`cc` [1] and `alto` [4]) and selected benchmarks and benchmark runs from the SPEC95 and SPEC2000 suites. Instead of following the traditional SPEC guidelines of training only with the designated "train" profiles and gathering performance statistics with the designated "reference" benchmark runs, we evaluate nearly all possible combinations of training and evaluation runs. We summarize the usefulness of basic block profiles in this wider context, evaluate the reliability of the results that we derived from using a range of evaluation runs, and evaluate the apparently uncontroversial claim that more accurate basic block profiles are connected to better profile-driven optimization performance. We find that while in the `alto` optimization context, there is a significant correlation between more accurate profiles and more useful profiles, no such correlation existed in the `cc` system.

1 Introduction

Profile-directed compiler optimization is a commonly implemented technique and is considered to be an effective one. Profile-directed optimization (PDO) depends on the assumption that having more accurate predictions of future behavior will result in better optimization performance. Both the assumption of effectiveness of profile-directed optimization, and the assumption of a connection between more accurate profiles and better profile-driven optimization performance, require quantification. In this paper, we attempt to evaluate both assumptions.

We present a methodology for evaluating the effectiveness of profile-directed optimization, for determining the significance of variability in profile-directed optimization performance and for measuring the strength of the connection of profile accuracy and profile "usefulness". We use this methodology to analyze profile-driven optimization for two optimization systems, `cc` and `alto`. Both profile-driven optimization systems used profile information to provide significantly better performance in the resulting code. However, we derive somewhat cautionary results concerning the commonly-held assumption that profile accuracy strongly predicts profile-directed optimization performation; there was only a weak connection between profile usefulness and accuracy for `alto` and none whatsoever for `cc`. Those wishing to discover whether these assumptions hold for their own optimizers, programs and profiling methods will need to repeat our analyses.

We evaluate metrics that attempt to measure how accurately a profile predicts a given future program execution. Suppose we have two training profiles p_1 and p_2 and an evaluation run (which when profiled, produces a profile p_E). Suppose also that the binary produced by using p_1 for profile-driven optimization performs better on that evaluation run than the binary produced by using p_2. A

*This research was supported, in part, by a grant from Intel MRL (Microprocessor Research Lab).

"good" profile comparison metric in this case would be one that shows that p_1 is a more accurate prediction of p_E than is p_2; that is, a metric that correlates strongly with profile usefulness.

We must immediately clarify the scope of this paper. We cannot derive results that apply to all profile-gathering techniques, benchmarks and optimizers. In this paper we work within the context of profiles that have been directly gathered by basic block profiling (as opposed to approximate methods such as statistical sampling, or profiles that are entirely synthetic, such as those generated by static estimation). We use two optimizers and a wide range of benchmarks but cannot generalize from these optimizers and benchmark programs to a hypothetical "universe" of benchmarks and optimizers.

Our methodology for analyzing profile-driven optimization performance and its relationship to accuracy is applicable to other optimizers, architectures, benchmark sets, and profiling methods. We feel that applying our methodology to the domain of exact basic-block profiles is the logical starting point for analysis of the relationship between profile usefulness and profile accuracy. The choice of which training profiles to use is more "fundamental" than the choice of profiling methods; regardless of what profiling methods are available, the issue of which training profiles to use will always be present.

2 Experimentation Framework

2.1 Definitions

In the process of profile-driven optimization, a given *run* (deterministic execution of a benchmark program with a certain input) produces a profile that is associated with that run. This profile is then used as input to a profile driven optimizer, and is thus called a *training profile*. The resulting binary can be evaluated with an *evaluation* run. The latter type of run will also have a profile associated with it, the *evaluation profile*, which is the basic block profile that would have results from profiling the binary with the evaluation run.

We draw our benchmarks from the SPEC95 and SPEC2000 benchmarks (if a benchmark exists in both benchmark sets, we use the SPEC2000 version and append "2000" to the benchmark name). The SPEC benchmarks define three standard runs, called `ref`, `test` and `train` (each of which can often be combinations of multiple program runs). The profile-driven optimizations allowed in the context of SPEC benchmarks involve using `train` as the training run and `ref` as the evaluation run (`test` may only be used for a relatively short-running test of the correctness of a given benchmarking setup). In our work, we use all of the available runs as training and evaluation runs, in all combinations. Where the SPEC benchmarks call for aggregating multiple runs into a single evaluation or training run, we consider each run individually. Thus, instead of testing a single training profile and evaluation run for profile-driven optimization, we may gather information on as many as 100 possible combinations of training profiles and evaluation runs (we use 10 different evaluation runs and 10 different training profiles for the SPEC2000 benchmark `perl` resulting in 100 possible combinations). More commonly, we have only the three standard SPEC runs available to us and thus gather information on 9 such combinations.

When presenting the names of non-standard SPEC runs (that is, runs that are not simply the SPEC training, testing or reference runs), we will indicate the source of the run as needed. There are two `perl2000` benchmark runs that involve calculation of "perfect" numbers. We refer to the run that is one of the multiple runs in the SPEC reference benchmark as `ref/perfect` and the run that was part of the SPEC training benchmark as `train/perfect`. Generally the names of these runs are not significant and are included only for reference.

We define *profile usefulness* in terms of an evaluation run. That is, it is meaningless to say that profile p_1 is more useful than p_2; only that p_1 is more useful than p_2 with respect to some evaluation run.

Profile accuracy, as measured by one of our profile comparison metrics, measures how well the behavior associated with a training profile predicts the behavior associated with an evaluation profile, strictly in terms of the contents of the two profiles. Once again, accuracy is defined in terms of an evaluation profile. The accuracy of profile p_1 (given a comparison metric) is calculated strictly by comparing the profile data associated with p_1 with the profile data associated with the evaluation run.

2.2 Profile-Driven Optimization Platform

We have implemented a system for evaluating profile usefulness and accuracy. This system consists of a set of profile gathering tools, a profile manipulation tool, and two optimization platforms (the `alto`[4]) system and the standard Digital Unix C Compiler[1]) using the profiles that we gather. The following steps outline the operation of our system.

First, we produce "base" binaries using the Digital Unix C compiler (DEC C V5.6, subsequently referred to as `cc`).

Second, we use `alto` to gather profile information and build a Control Flow Graph (CFG). The base binaries are instrumented by `alto` and used to gather profile information for the various runs of the benchmark.

Third, these profiles and the benchmark's control-flow graph are passed to the profile manipulation tool, which may apply transformations to real profiles or generate new profiles from scratch. The profile optimization tool can generate profiles in `alto` format or in the standard `pixie` format. At this stage we also gather data on profile characteristics and comparisons between profiles.

Fourth, these new profiles are used as inputs to the profile-driven optimization process. These profiles are used with either `alto` (with full optimizations switched on) or the Digital C compiler (see [1] for details of the optimizations performed) to produce an optimized binary. The profile-driven optimizations that provide the most substantial improvements are similar in both optimzers: code placement optimizations, procedure inlining, and super-block formation (profile-driven optimization steps in super-block formation also affect many subsequent optimizations that are not themselves profile-driven). We also produce binaries with the same set of optimization flags but without profile information, for comparison.

Finally, the optimized binary is run[1]. We can compute cycle counts (using the EV5 performance counters) for all our evaluation runs at this time. We are often measuring only subtly different binaries, with very small variations in run-time. We run our benchmarks on a 333Mhz EV5 21164 machine with 1GB of memory (running Digital UNIX V4.0). The machine, while old, has highly accurate performance counters and mature and well-tuned optimizers.

Our work is not focused on producing peak optimization performance. Our focus is on studying the effects of profile-driven optimization and methods for evaluating its effectiveness, not implementing the fastest possible optimizations. In general, the optimization performance of our system (through either the `alto` path or the `cc` path) is good. Usually, optimization performance is within 5-10% of the DEC C compiler at the highest level of optimization, and sometimes faster, due mainly to the aggressive whole program optimizations implemented in `alto`.

We use the technique of using the evaluation profile as a training profile, a case that we call, after Savari and Young [5], *resubstitution*. While not valid as a practical technique (why run the exact same program execution twice?), resubstitution frequently generates interesting results, allowing us a insight into how much benefit results from having "perfect" information. We do not use resubsitution cases when reporting average benefits from using profile-directed optimization.

Our goal is to investigate the usefulness and accuracy of profiles, not to generate superior SPEC results or to find the ideal "representative" training profile. Our use of non-standard SPEC training profiles and evaluation runs means that our results cannot be considered to be valid SPEC results. This does not render the results invalid in a research sense. As stated above, even the (highly questionable in a benchmarking sense) use of resubstitution can generate interesting data. We carry out analyses to determine whether our observed performance effects from shorter-running evaluation runs than the SPEC "ref" benchmarks represent real effects or whether the effects are simply due to experimental error; the former is true for nearly all combinations of optimizer, benchmark and evaluation run.

[1]Currently, we have some missing data points (including the SPEC2000 version of gcc) due to bugs in one or the other of the optimizers, including a number of the baseline "non-profile-directed optimization case" results. We are also missing some entire benchmarks in the `cc` optimization context. Our results are not significantly altered by restricting the benchmark sets to only those benchmarks that worked across both optimization environments, so we have opted to present more information (the benchmarks that worked only under the `alto` environment) rather than less.

3 Results

3.1 Usefulness of Profile-Directed Optimization

We gathered cycle counts for each combination of optimizer, benchmark, training profile and evaluation run. We repeated each evaluation run 11 times, discarding the first cycle count score due to significant differences in the first run (almost certainly due to page faults as the program binary is brought into memory from disk). We calculated average cycle counts from the other 10 evaluation runs. We present these average cycle counts normalized by the average cycle counts of the comparison binaries; that is, the optimized binaries that did not used profile-directed optimization. Thus, for a given evaluation run, a binary produced by profile-directed optimization that runs 5% faster than the binary produced by non-profile directed optimization is assigned a score of 0.95 in Table 1.

In Table 1, we present results showing the relative performance of profile-directed optimization for our different benchmarks as compared to the same benchmarks optimized without profile directed optimization. As each benchmark has multiple evaluation and training runs, we present the average profile-driven optimization performance for all of the combinations of evaluation and training runs, excluding the "resubstitution" case.

Overall, profile-directed optimization is an effective technique (an average improvement of 3%), but the results are sharply variable: there are several benchmarks where all training profiles make the program slower for each evaluation run. A majority of benchmarks for both optimizers have at least one combination of training profile and evaluation run where profile-directed optimization performs badly.

Examining the individual benchmark runs, we observe a wide range of performance variability. Table 2 presents the top and bottom benchmark runs by profile-driven optimization variability. There is a huge range of variability among evaluation runs.

Given that cycle counts have a degree of variability due to experimental error, we used a simple technique (one-way ANOVA[2] [8]) to determine whether, for each evaluation run, the differences between cycle counts from binaries trained on different training profiles were significant. That is, were we observing real differences between training runs or were the differences that we observed entirely due to experimental error? This issue is somewhat more pressing for this work than it is for more conventional profile-driven optimization research, as some of the runs which we were using as evaluation runs were comparatively brief (as compared to the standard SPEC "`ref`" runs). The one-way ANOVA procedure ("one-way" because we vary only a single variable; "ANOVA" is short for "ANalysis Of VAriance") attempts to determine, given a set of experimental results gathered at different 'levels' (in this case, using different training profiles), whether there are statistically significant differences among the results for different levels. That is, we attempt to disprove the null hypothesis that the average cycle counts for a given evaluation run are the same regardless of which training profile was used. If the probability that this could be the case is sufficiently low, we can reject this null hypothesis and conclude that in fact there are statistically significant differences between the profile-driven optimization effects of different training profiles.

We were able to reject the null hypothesis of "no significant difference exists between the effect of training profiles" at a significance level of 0.05 (that is, we found that it is no more than 5% likely that, given no effect at all from training profiles, we would have seen the pattern of variability that we did) for all but 5 evaluation runs (4 under `alto` - two runs in the `art` SPEC2000 benchmark and one run each for `compress` and `parser`, 1 under `cc` - one run under `compress`). For the vast majority of our benchmark runs, the probability that we would have observed the variability that we did due to to factors other than the training profile is negligible (under 0.001).

[2] The results for one-way ANOVA are far too verbose to present here, and many of the details are beyond the scope of this paper. One-way ANOVA merely detects that there exists some significant difference between at least one training profile and the rest - it does not in itself yield results analyzing how many of the training profiles differ significantly from the others. Thus, the results of a one-way ANOVA should be treated with a degree of caution - when we say that a significant difference exists for some `perl2000` evaluation run with 10 different training profiles, we are only allowed to say that "some difference exists among the usefulness of those 10 profiles", as opposed to the stronger statement "each and every one of these profiles is significantly different from every other one" or any of the intermediate possibilities. We carried out post-hoc analyses to distinguish between this set of posibilities, but the details are again beyond the scope of the paper.

Optimizer	Benchmark	Number of runs	Normalized execution time		
			Minimum	Maximum	Mean
alto	ammp	3	0.97	0.98	0.98
	bzip2	5	0.87	1.01	0.93
	compress	3	0.94	1.06	0.99
	crafty	3	0.89	0.93	0.91
	gap	3	0.95	0.97	0.95
	go	5	0.96	1.06	0.99
	gzip	7	1.00	1.14	1.06
	ijpeg	3	0.96	0.98	0.97
	li	3	0.97	0.99	0.98
	m88ksim	3	0.83	1.00	0.89
	mcf	3	1.00	1.02	1.01
	parser	3	1.00	1.02	1.01
	perl2000	10	0.83	1.08	0.96
	twolf	3	0.93	1.01	0.97
	vortex2000	5	0.86	0.91	0.89
	ALL CASES		0.83	1.14	0.97
cc	ammp	3	0.99	1.04	1.02
	bzip2	5	0.91	1.06	0.96
	compress	3	0.92	1.02	0.99
	crafty	3	0.94	0.98	0.96
	equake	3	0.95	1.01	0.99
	gap	3	0.92	0.99	0.95
	go	5	0.99	1.14	1.06
	ijpeg	3	0.94	0.98	0.96
	li	3	0.84	0.92	0.87
	m88ksim	3	0.88	1.07	0.96
	mcf	3	0.99	1.00	1.00
	perl2000	10	0.86	1.13	1.00
	twolf	3	0.93	0.98	0.95
	vortex2000	5	0.90	0.99	0.94
	ALL CASES		0.84	1.14	0.97

Table 1: Execution time of PDO binaries over all evaluation runs and training profiles (each set of evaluation run results normalized such that the non-profile-directed optimization case is equal to 1.0 for each evaluation run).

Optimizer	Benchmark	Evaluation run	Fastest Case	Slowest case	Mean	Standard Deviation
alto	perl2000	train/diffmail	0.90	1.05	0.96	0.0457
alto	perl2000	ref/diffmail	0.90	1.06	0.96	0.0457
alto	perl2000	ref/perfect	0.80	0.96	0.89	0.0446
cc	perl2000	train/scrabble	0.82	1.00	0.93	0.0433
cc	go	ref2	1.00	1.12	1.08	0.0427
cc	go	train	1.01	1.14	1.08	0.0426
cc	go	test	1.00	1.12	1.08	0.0412
alto	perl2000	ref/makerand	0.78	0.93	0.87	0.0408
...	...	...	...	...	...	...
alto	gzip	program	1.13	1.14	1.13	0.0021
alto	parser	ref	1.00	1.00	1.00	0.0020
alto	ijpeg	train	0.98	0.98	0.98	0.0013
alto	ammp	train	0.98	0.98	0.98	0.0010
cc	mcf	ref	1.00	1.00	1.00	0.0009
alto	parser	train	1.01	1.01	1.01	0.0008
alto	ammp	ref	0.98	0.98	0.98	0.0006

Table 2: Evaluation runs with highest and lowest variability due to profile-directed optimization profile choice; units are normalized as for Table 1.

3.2 Connection of Usefulness and Accuracy

3.2.1 Profile Accuracy Metrics

All of our comparison metrics compare a list of basic block counts in a training profile with a list of basic block counts in an evaluation profile. They return a single number, a score that indicates how well the basic block counts in the training profile predict the basic block counts in the evaluation profile. Thus, a more accurate training profile better predicts the CFG-level behavior of the evaluation run. Most of these metrics are asymmetric.

A profile comparison metric consists of a comparison type and a way of applying it over the program. The comparison types we use in this paper are key-matching, static coverage and relative entropy.

Key-matching is introduced in [7]. It uses a parameter that determines how many blocks are selected for key-matching. For example, if a function has 50 blocks, and the matching level is 40% (or 0.4), then we perform key-matching on the top 20 blocks as follows: the key-match score is the number of blocks in the top 20 blocks in the training profile that are also in the top 20 of the evaluation profile. Key-matching metrics are denoted by KM(level) - "level" is always 0.1 in this paper.

Static coverage (denoted "STCOV") measures what proportion of the blocks executed ("covered") in the evaluation profile are also executed in the training profile.

Relative entropy (denoted "ENT") as a method of comparing profiles was introduced by Savari and Young [5] and is fully described there. Relative entropy treats the profiles being compared as distributions of random variables and uses an information-theoretic approach to measure the difference between the two distributions.

We use two methods for applying these comparisons to our programs. Firstly, we can apply the comparisions to the whole program's set of basic block counts directly. This is the default method. Secondly, we can apply them only to the entry counts of functions, ignoring all other basic block data (denoted by prefixing "FE-" to the comparison name in our results).

3.2.2 Evaluating the Connection Between Comparison Metrics and Usefulness

To measure the association between profile usefulness and a given profile comparison metric, we use the Spearman Rank Correlation Coefficient [8], r_s. r_s can be calculated by assigning ranks to the values being compared (scoring

ties as the average rank values - so if there is a tie between the top two values, they both are assigned the rank of 1.5) and calculating the more familiar Pearson correlation coefficient [8] over those ranks. Thus, calculations of r_s discard the magnitude of the differences between data points. This makes r_s weaker (more likely to miss a real effect) than Pearson's correlation coefficient but much more robust in the presence of non-linear relationships, outliers and (more generally) data that does not hold to a bi-variate normal distribution.

When analyzing the correlation between profile accuracy and usefulness, we must be aware that there is no "natural" population of profiles for a given benchmark. For most benchmarks, we have a limited number of runs available to us, and they have been chosen artificially. If we include other profile types besides profiles derived directly from real runs, we are introducing further artificial biases into our population. Admittedly, the choice of benchmark runs from the SPEC benchmark sets are artificial also, but they are not the artificial choices of the authors of this paper - that is, they are not hand-picked to advance our favored hypotheses.

We will proceed to show an example of how we evaluate the connection between profile usefulness and accuracy. Firstly, we present the average cycle count scores and usefulness scores for the benchmark `perl2000` and the `ref/perfect` benchmark evaluation run. For each training profile, we have an average cycle count (reflecting how many cycles the binary that was produced by profile-driven optimization using that profile took to run the evaluation run) and an accuracy score (reflecting how close the training profile was to the profile produced by the evaluation run). For this example, we will use the accuracy scores provided by relative entropy[3].

Table 3 shows the cycle counts and relative entropy scores for a list of training runs (the names refer to the different benchmark runs available for `perl2000` and are not of any interest aside from the fact that they label cases). To calculate a score for how closely relative entropy predicts scaled cycle counts, we take the r_s value of two variables (cycle count and relative entropy) over the list of cases (training profiles), which turns out to be $r_s = 0.87$. This value is statistically significant at the 0.01 level; that is, if there was no association whatsoever between two variables, we'd expect to see a r_s value this high less than 1 in 100 times. In fact, the chance that we would see such a strong association between two unconnected variables in such a list of cases is less than 1 in 1700. The proportion of scaled cycle count variation explained by relative entropy is $r_s^2 = 0.75$ - that is, 75% of the variation in average profile-driven optimization performance in this particular case can be explained in terms of relative entropy.

Training run	Cycle count (GCycles)	Relative entropy
`ref/diffmail`	45.819	8.05
`ref/makerand`	47.581	22.57
`ref/perfect`	40.774	0
`splitmail1`	46.495	8.52
`splitmail2`	45.640	8.20
`splitmail3`	47.176	8.35
`splitmail4`	45.281	8.29
`train/diffmail`	45.615	8.06
`train/perfect`	42.515	2.45
`train/scrabble`	48.923	20.44

Table 3: Example 1: `perl2000` scaled cycle counts and accuracy metrics for a single evaluation run (`ref/perfect`)

Note that this benchmark has a quite large number of possible training profiles (10). Many of our benchmarks have only 3 or 4 runs available, so we are often in the situation of calculating correlations over a tiny set of cases. In this circumstance, it is possible to have apparently strong correlations that are in fact statistically meaningless *on their own* . Only when they occur as a pattern across multiple evaluation runs and/or benchmarks can we attach any weight to these results.

Table 4 shows this analysis repeated for all of our evaluation runs in `perl2000`. We will see a larger set of results - now, we have a table with r_s numbers for each evaluation run. Not all of the correlations are significant at a 0.01 level (those that are are marked with a "**") or even at a 0.05 level (marked with a "*"). For example, the value $r_s = 0.382$, seen for the evaluation run `ref/diffmail` is fairly low: there is a 14% chance that two unconnected variables might show a rank correlation equal to or greater than this value (3 evaluation runs fall into the category of

[3]More accurate profiles produce lower relative entropy scores, zero represents a perfect match

Evaluation run name	r_s score
ref/diffmail	.382
ref/makerand	.778**
ref/perfect	.867**
splitmail1	.697*
splitmail2	.612*
splitmail3	.685*
splitmail4	.612*
train/diffmail	.394
train/scrabble	.285

Table 4: Example 2: All perl2000 evaluation runs with the rank-correlation values of cycle counts and relative entropy calculated over each training run

not being significant at the 0.05 level). However, even considering only these three values in isolation, it is unlikely that we would see three such correlations (that is, positive and in the range $0.285 < r_s < 0.394$) between relative entropy and average cycle count if overall, there was no connection between relative entropy and average cycle count for any of these runs. In fact, the chance that such three correlations this strong or stronger would have arisen by chance given no connection between relative entropy and cycle count is about 1%.

Note that it is quite possible to have negative r_s scores; in this case, more accurate profiles actually result in worse profile-driven optimization performance.

We can compute a summary value for the overall connection of usefulness and accuracy over a benchmark by averaging the r_s values for each evaluation run, yielding an aggregate correlation of $mean(r_s) = 0.59$ for the perl2000 benchmark.[4]

Using such a procedure to gather aggregate numbers for each benchmark, this time over a range of comparison metrics, we derive Table 5. This table shows the aggregate r_s scores for each comparison, benchmark and optimizer, as well as overall mean scores for r_s comparison and optimizer. It is clear that the perl2000 benchmark, presented above, and particularly the perfect evaluation run, represent a quite favorable case - note the large number of benchmarks in this table for which the aggregate r_s scores are either very low (i.e. no correlation) or actually negative (i.e. more accurate profiles have worse profile-driven optimization performance). Particularly, the results for the cc optimizer show no overall pattern of a connection between profile usefulness and profile accuracy.

In the alto case, all of the profile comparison metrics yielded small but significant correlations between profile accuracy scores and profile usefulness scores. Key-matching performed slightly worse than the other two profile accuracy metrics, entropy and static coverage. The "function-entry" versions of these latter accuracy metrics performed slightly better than the versions that considered all of the basic blocks in the program, although such a small difference is not likely to be significant.

There was a substantial amount of variability among the aggregate r_s scores for each benchmark. Some of this variability is simply random; the aggregate r_s scores for the benchmarks with a small number of runs are subject to a great deal of randomness as they involve comparisons among only 9 or 16 values. However, some benchmarks clearly have far stronger associations between usefulness and accuracy than others. Recall that the correlation coefficients in this table rank how well profile usefulness correlates with profile accuracy; they have nothing to say about how well profile-directed optimization works overall.

A major weakness of the above approach to evaluating the connection of profile-directed optimization performance and profile accuracy is that, due to the use of non-parametric methods and averaging across different benchmarks, small variations in one benchmark are weighted as heavily as huge variations in another. There is no simple way to avoid this problem without recourse to parametric correlation methods. However, we can derive results that are more useful by restricting our above analyses to only those evaluation runs with greater variability due to profile-directed optimization. The overall (per-optimizer) results from restricting our analysis to the top half of evaluation runs with the highest level of profile-directed optimization variability are shown in Table 6.

The failure of our profile accuracy metrics to explain

[4]This is not generally good practice; more statistically rigorous is to transform each r_s value to a z-score (normal score), take the average over these z-scores and transform back into the range of r_s. However, this procedure is complex and results in average r_s scores little different from those that we derive from simple averaging. Similarly, we will not present significance results for aggregate r_s scores here; the statistical justification for these results in beyond the scope of this paper.

Optimizer	Benchmark	$mean(r_s)$					
		ENT	STC	KM(0.1)	FE-ENT	FE-STC	FE-KM(0.1)
alto	ammp	-0.67	-0.79	-0.50	-0.50	-0.58	-0.67
	art	0.20	0.23	0.35	0.25	0.07	0.23
	bzip2	-0.12	-0.06	0.09	0.00	0.14	0.16
	compress	1.00	0.91	0.83	1.00	0.29	0.58
	crafty	0.83	0.67	0.67	0.67	0.17	0.50
	equake	0.17	0.17	0.00	0.17	0.00	0.58
	gap	0.83	0.83	0.50	0.83	0.79	0.50
	go	0.26	0.13	0.20	0.34	0.30	0.23
	gzip	-0.16	-0.24	-0.28	-0.26	0.12	-0.24
	ijpeg	-0.83	-0.67	-0.67	-0.83	0.00	-0.79
	li	0.83	0.96	0.50	0.83	0.96	0.50
	m88ksim	0.67	0.67	0.67	0.67	0.79	0.67
	mcf	0.50	0.62	0.67	0.50	0.58	0.87
	parser	-0.50	-0.83	-0.83	-0.50	-0.29	-0.67
	perl2000	0.59	0.60	0.45	0.52	0.60	0.50
	twolf	0.33	0.33	0.17	0.33	0.58	0.29
	vortex2000	0.38	0.36	0.38	0.64	0.17	0.48
	vpr	0.52	0.59	0.57	0.55	0.51	0.54
	alto MEAN	0.27	0.25	0.21	0.29	0.29	0.24
cc	ammp	0.00	-0.04	0.33	0.33	-0.58	0.00
	bzip2	0.16	0.04	0.15	0.06	-0.07	-0.09
	compress	0.33	0.17	0.17	0.33	0.29	0.29
	crafty	0.67	0.33	0.00	0.33	-0.46	0.00
	equake	-0.83	-0.83	-0.50	-0.50	0.00	0.00
	gap	-0.33	-0.33	-0.33	-0.33	-0.46	-0.17
	go	-0.72	-0.76	-0.74	-0.72	-0.60	-0.65
	ijpeg	0.33	0.00	0.00	-0.33	0.00	-0.33
	li	-0.17	-0.46	0.33	-0.17	-0.46	0.33
	m88ksim	-0.17	-0.17	-0.17	-0.33	-0.12	-0.17
	mcf	0.33	0.33	0.17	0.33	0.58	0.29
	perl2000	-0.18	-0.23	-0.17	-0.19	-0.21	-0.14
	twolf	0.33	0.33	0.17	0.33	0.58	0.33
	vortex2000	0.04	-0.01	-0.02	0.06	0.02	0.06
	cc MEAN	-0.06	-0.12	-0.04	-0.06	-0.11	-0.02

Table 5: The connection of usefulness and accuracy: aggregated r_s scores over optimizers, benchmarks and different comparison metrics

Optimizer	ENT	STC	KM(0.1)	FE-ENT	FE-STC	FE-KM(0.1)
alto mean	0.57	0.52	0.48	0.58	0.45	0.45
cc mean	-0.15	-0.22	-0.14	-0.17	-0.03	-0.03

Table 6: Aggregated r_s scores over optimizers, considering only the top half of evaluation runs by PDO variability

`cc` profile-directed optimization performance turns out to be unconnected to profile-directed optimization variability. Even considering only benchmarks and benchmark runs that had large variations in profile-directed optimization performance did not improve the connection between profile accuracy and profile usefulness when using `cc`. However, our `alto` results become substantially stronger when we eliminate benchmark runs with small variations in profile usefulness. Entropy-based methods, in particular, improve markedly. The "FE-ENT" accuracy metric predicts 34% of the variation in our profile-directed optimization results under `alto` - a modest result, but the strongest one so far.

We found no similar improvements from restricting our analysis to smaller (e.g. top quarter by PDO variability) subgroups of our evaluation runs. Not suprisingly, the bottom half of evaluation runs by PDO variability showed no significant correlation (under `alto` or `cc`) between profile accuracy and profile usefulness.

3.2.3 Discussion

There was no reason to suppose that any reliable connection between accuracy and usefulness existed in the `cc` optimization context whatsoever. We conjecture that the much more extensive and high-level optimizations present in `cc` sufficiently transform the control-flow-graph to the point where the relatively subtle differences between training profiles are irrelevant. This does not mean that profile-driven optimization does not work in `cc`, nor does it mean that arbitarily inaccurate profiles will produce profile-driven optimization performance indistinguishable from good ones. What it does mean is that, within the fairly narrow range of profiles and benchmarks we tested, accuracy could not be shown to have any connection to usefulness. We evaluated many other profile comparison metrics than (carrying out key- and weight-matching at multiple levels, using dynamic coverage) presented here and found that none of them performed any better than the comparison metrics presented.

Our results for the `alto` optimization context were more encouraging, but still relatively weak. Even when restricting our analysis to benchmarks with large profile-directed optimization variability, we could explain no more than a third of the variation in average cycle counts by some accuracy metric.

One of the most startling results was the fact that the accuracy metric "FE-STC" performed as well as it did despite the fact that it ignores away nearly all of the information in the block profile. This extremely simple metric can be calculated by determining the number of functions entered in the training profile and the evaluation run divided by the total number of functions entered in the evaluation run.

The effectiveness of this metric (and similarly restricted metrics) could result from there being little variation in within-function behavior from run to run (that is, when profiles produced from benchmark runs differ, it is because they cover a different set of functions, not because they have radically different behavior within those functions). An alternate possibility is that the optimizations in `alto` really only effectively worked at a per-function level and thus made little use of the within-block information (code placement optimizations that work at a whole-function level and procedure inlining are both examples of optimizations that work very well with only per-function information, although both benefit from knowledge of call site counts - or, nearly equivalently, call graph edge counts). These possibilities are not easily separated, although the fact that our "function-entry only" comparison metrics are strongly correlated ($r_s > 0.9$) with their whole-program counterparts for nearly all benchmarks is suggestive that the former possibility is true (across both optimizers, `bzip2` and `gzip` were the only exceptions).

4 Related Work

Wall [7] makes the first systematic attempt to evaluate profile accuracy. Wall compares real profiles and static estimates for accuracy using key- and weight-matching to compare profiles. His comparisons use key- and weight-matching at both fixed levels (top k) and, similar to our work, at levels proportional to the total number of blocks (top $N\%$). He shows strong improvements in accuracy from using real profiles over static estimates. He briefly analyzes some theoretical optimization algorithms, showing weaker results, and warns against unrealistic expectations concerning profile driven optimization. Wu and Larus discuss static estimation in [9], using Dempster-Shafer theory to combine branch prediction heuristics. Key- and weight-matching are used to evaluate the accu-

racy of the static profiling methods. Wagner et. al do a similar analysis to Wall's in [6].

These works do not attempt to establish any connection between profile accuracy and profile-driven optimization performance. Our work diverges from all of these works by connecting accuracy metrics to actual profile-driven optimization performance in two mature optimizers.

Fisher and Freudenberger report that profile data gathered from previous runs yields good branch predictions [3]. They mention the possibility that the differences in real benchmark runs might be related to the benchmark's coverage of the program as opposed to differences in behavior in code that is covered by both runs. This is an interesting observation, which unfortunately they were not able to quantify. Our results suggest that this intuition was correct (at least in terms of what information `alto` was able to use effectively); the comparatively strong predictive value of the accuracy metric "FE-STC" (function entry static coverage) supports this.

An extensive treatment of information-theoretic methods for comparing and combining profiles, including the relative entropy comparison used in this work, appears in Savari and Young [5]. Our work validates the use of relative entropy as a profile comparison metric.

Cohn and Lowney compare the differences in usefulness between profile-driven optimization and static estimation on the Compaq Alpha in [1]. They report a substantial speedup (17%) on the SPEC 95 integer benchmarks from using feedback directed optimization. Their results show a larger effect from profile-driven optimization than this paper; they use more aggressive optimizations on a more recent iteration of the Alpha architecture. Another difference between their work and ours is that we use a wider variety of benchmarks (including SPEC2000 and floating point benchmarks) and benchmark runs than they do; this may also contribute to the performance gap between this paper and their work.

Eeckhout et al. [2] use statistical data analysis techniques to cluster similar "program-input pairs" (in our terms, pairs consisting of a benchmark and an evaluation run). They concentrate on overall benchmark characteristics as opposed to profile accuracy and/or profile usefulness.For our analyses in this paper, we have little need to reduce the number of "program-input pairs" to cover a hopefully representative set of benchmarks, training profiles, and evaluation runs, as our analyses benefit from more data points rather than fewer. This is true even if some of the training profiles and evaluation runs produce very similar effects.

5 Conclusion

Profile-directed optimization is a worthwhile technique, on average, in both of the optimizers evaluated. On average, we saw an improvement over non-profile-directed optimizations of about 3.5% on `alto` and 5% on cc; these aggregate numbers concealed substantial variations (the best case for either optimizer was approximately 17% better than non-profile-directed optimization and the worst case for either was approximately 14% worse).

Nearly all of the benchmark runs showed significant variation in profile-directed optimization performance. In only 1% of our evaluation runs were we unable to detect significant variation among profile-directed optimization performance (that is, no variation due to profile-directed optimization existed or it was so small that we were unable to separate this variation from experimental error). Again, large differences existed between the evaluation runs with the largest amount of profile-directed optimization variability and those with the smallest - the standard deviations in speed-up over the non-profile-directed-optimization case ranged from effectively zero to nearly 5%.

Profile accuracy is only weakly associated with profile usefulness in one of our optimizers (`alto`) and not connected at all with profile usefulness in another (`cc`), for our set of benchmarks and benchmark runs. While considering only benchmarks or runs with higher variability in profile-driven optimization performance improved the connection on `alto`, the connection between usefulness and accuracy still accounted for only 34% of the observed variation in profile-driven optimization performance. While the comparatively weak (non-parametric) correlation methods that we had to use may have caused us to be overly conservative, it seems unlikely that any accuracy metric whatsoever would explain in excess of 50% of the variation. Of the variation in profile usefulness explainable by profile accuracy metrics, much of it was explainable by fairly simple profile accuracy metrics, most notably static coverage of function entries ("FE-STC"). We find some quantitative support for Fisher and Freuden-

berger's claim [3] that differences in exact profiles are mainly due to a different set of functions being covered in different runs, as opposed to different behavior within the functions from run to run.

That the overall results are negative for `cc` and weak for `alto` is not entirely suprising. Much of the variation in our training profiles does not necessarily cause different optimization outcomes. That which does does not necessarily help. Not every optimization "decision" produces better performance, regardless of whether it is based on good information - few compiler optimizations are truly "optimizations", particularly when interacting with many other optimizations. We see substantial and significant variations due to profile choice in profile-driven optimization, and for most benchmarks, much of this variation is not explainable in terms of profile accuracy. This suggests that there is a large component of randomness in the outcome of the profile-driven optimization process.

Our major contributions are twofold. Firstly, we have developed a methodology for evaluation of profile-driven optimization performance and its connection to profile accuracy that can be applied to any combination of processor architecture, optimizer, and set of benchmarks. Secondly, our results show that there exists at least one optimizer for which usefulness and accuracy are not correlated (in our experimental context) and one in which this correlation exists but fails to explain the bulk of profile-directed optimization performance.

Therefore, any claims about profile-directed optimization techniques or more accurate profiling techniques (or the necessity of obtaining more accurate precise basic block profiles - dynamically or otherwise) should be evaluated *experimentally*, not in terms of profile accuracy. We have shown that there are a range of cases where little or no connection between profile accuracy and profile usefulness exists. Thus, it is incumbent on designers of profile-directed optimization systems to demonstrate that the profile-directed optimizations in their systems are actually effective over a wide range of benchmarks, rather than merely showing that the profiles gathered are of high accuracy.

References

[1] R. Cohn and P. Lowney. Feedback directed optimization in Compaq's compilation tools for Alpha. In *In Proc. 2nd Workshop on Feedback Directed Optimization, 1999*, 1999.

[2] L. Eeckhout, H. Vandierendonck, and K. D. Bosschere. Workload design: Selecting representative program-input pairs. In *The Eleventh International Conference on Parallel Architectures and Compilation Techniques (PACT-2002)*, 2002.

[3] J. Fisher and S. Freudenberger. Predicting conditional branches from previous runs of a program. *Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 85–95, 1992.

[4] R. Muth, S. Debray, S. Watterson, and K. de Bosschere. alto: A link-time optimizer for the DEC Alpha. Technical Report TR98-14, Department of Computer Science, The University of Arizona, 1998.

[5] S. Savari and C. Young. Comparing and combining profiles. In *Proc. Second Workshop on Feedback-Directed Optimization (FDO)*, 1999.

[6] T. A. Wagner, V. Maverick, S. L. Graham, and M. A. Harrison. Accurate static estimators for program optimization. *ACM SIGPLAN Notices*, 29(6):85–96, 1994.

[7] D. W. Wall. Predicting program behavior using real or estimated profiles. 26(6):59–70, June 1991. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*.

[8] R. Walpole, R. Myers, and S. Myers. *Probability and Statistics for Engineers and Scientists*. Prentice Hall, 1998.

[9] Y. Wu and J. Larus. Static branch frequency and program profile analysis. In *In 27th International Symposium on Microarchitecture*, pages 1–11, 1994.

Practical Selective Replay for Reduced-Tag Schedulers

Dan Ernst and Todd Austin
Advanced Computer Architecture Lab
University of Michigan
Ann Arbor, MI 48109
{ernstd, austin}@eecs.umich.edu

Abstract

The trend towards deeper microprocessor pipelines has made it advantageous or necessary to predict the events that may happen in the stages ahead. A widely-used example of this technique is latency speculation, where the non-deterministic latency of some instructions, such as loads, forces dependents to predict the number of clock cycles these operations will take to complete execution. If there is a misprediction, those dependents that issued speculatively must be restarted or delayed appropriately so that they can execute again with the correct inputs. This process is called a scheduler replay. In the interest of reducing the replay penalty, some recent designs, such as the Pentium 4, have adopted selective replay mechanisms, which reschedule only data-dependent instructions on a latency misspeculation.

The deep pipelining trend has also forced designers to reduce the circuit complexity of individual stages to maintain high clock speeds and to keep power dissipation manageable. Tag elimination [4] is a technique used to reduce the complexity of a processor's issue stage by designing for the average case instruction. In Kim and Lipasti's recent work on Half-Price architectures [9]*, the authors state that it is "impractical" to implement a selective replay mechanism in a machine that uses tag elimination. In this paper, we detail the implementation of a practical selective replay method that is compatible with tag elimination schedulers and discuss the power and performance trade-offs that should be considered when designing a replay system.*

1 Introduction

Recent microprocessor designs have employed increasingly deep pipelines. Breaking the execution core into smaller pieces allows for higher clock speeds and, as a result, higher instruction throughput. Many recent studies [6][8][17] show that more benefit could still be extracted by this technique, indicating that pipelines will likely continue growing longer in the future.

However, the benefits of deep processor pipelines do not come without drawbacks. Placing extra stages in certain segments of the processor may force earlier stages to speculate on the events that may occur later in the pipeline [2]. For example, instruction latency speculation predicts how many cycles a producer instruction will take to complete its execution, which occurs several stages later in the pipeline. This is done so that consumer instructions can be issued to meet their inputs at the optimal time. If the prediction is wrong, however, at least some of the instructions in the stages between issue and execution must be restarted or delayed.

In the interest of keeping the latency misprediction penalty to a minimum, some processor designs, such as the Pentium 4, have included implementations of selective instruction replay. By using this technique, a processor only needs to re-execute instructions which are data-dependent on the misspeculated instruction.

Another pipelining obstacle is that some stages, such as the instruction scheduler, are often too big and slow to fit in a single cycle, and are also particularly resistant to the decomposition process necessary to implement pipelining [14].

Tag elimination [4] was proposed as a solution to an instruction scheduler complexity problem. By tailoring the reservation station structures for the common case of instructions needing to wait for fewer than two inputs, the scheduler tag bus capacitance can be reduced by up to 75%, allowing for a higher clock rate and lower power consumption. In Kim and Lipasti's paper "Half-Price Architecture" [9], the authors introduce a clever selective replay implementation that performs parent-child dependence propagation using the scheduler broadcast busses. In addition, they correctly point out that the combination of tag elimination with broadcast-based selective instruction replay is not practical to implement.

In this paper, we describe selective replay methods that are compatible with tag elimination. Furthermore, we analyze the performance and power consequences of the replay implementation decision.

The remainder of this paper is organized as follows. Section 2 gives background information on tag elimination. In Section 3, we present the different replay mechanisms that are then analyzed in Section 4. Related work is listed in Section 5 and our conclusions are presented in Section 6.

2 Tag Elimination

The techniques presented in [4] draw from the observation that most scheduler tag comparisons are superfluous to the correct operation of the instruction scheduler. Analyses reveal that most instructions placed into the instruction window do not require two source tag compar-

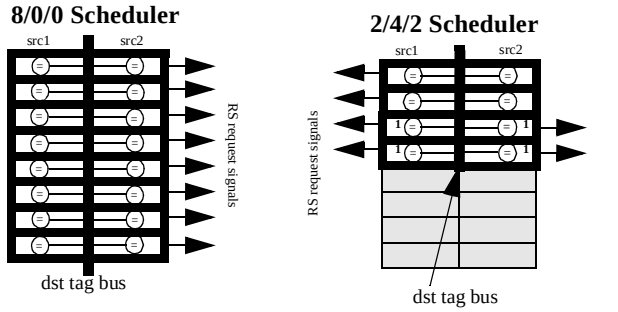


Figure 1. Conventional and Reduced-Tag Reservation Stations. The circles represent tag comparators. The bold tag entries include a comparator. The shaded tag entries are not necessary and thus do not include comparators. Entries with a '1' hold instructions with only one tag to compare against.

ators because one or more operands are ready, or the operation doesn't require two register operands.

Two scheduler tag reduction techniques were proposed that work together to improve the performance of dynamic scheduling, while at the same time reducing power requirements. First, a reduced-tag scheduler design was proposed that assigns instructions to reservation stations with two, one, or zero tag comparators, depending on the number of input operands in flight. An example of a scheduler window using tag elimination is shown in Figure 1.

To reduce tag comparison requirements for instructions with multiple operands in flight, we also introduced a last-tag speculation technique. This approach predicts which input operand of an instruction will arrive last and schedules the execution of that instruction based solely on the arrival of the final operand. Since the earlier arriving tags do not precipitate execution of the instruction, the scheduler can safely eliminate the comparator logic for all but the last arriving operand.

Because scheduling windows that use last-tag prediction don't track the readiness of the operands that are predicted to arrive earlier, a small table must be placed in the following pipeline stage to check that the prediction was correct and all operands actually arrived. This table simply consists of one ready bit for each physical register that is set when values become ready.

3 Implementing Selective Replay

3.1 Parent-Child Broadcast

In the Half-Price Architecture paper [9], Kim and Lipasti present one possible implementation of selective replay. An illustration of this scheme is shown in Figure 2.

Along with its input tags, an instruction's dependence information is kept in the instruction window in the form of one dependence matrix for each input operand. This matrix consists of *W* x *D* bits, where *W* is the machine width, and *D* is the depth of the *load shadow* [2], which is defined as the number of stages between instruction issue and notification of a cache hit or miss. In each matrix position, the presence of a "1" indicates that an instruction is in the corresponding slot in the scheduler pipeline that the current instruction is dependent on, either directly or through some intermediate instructions. Every cycle, the

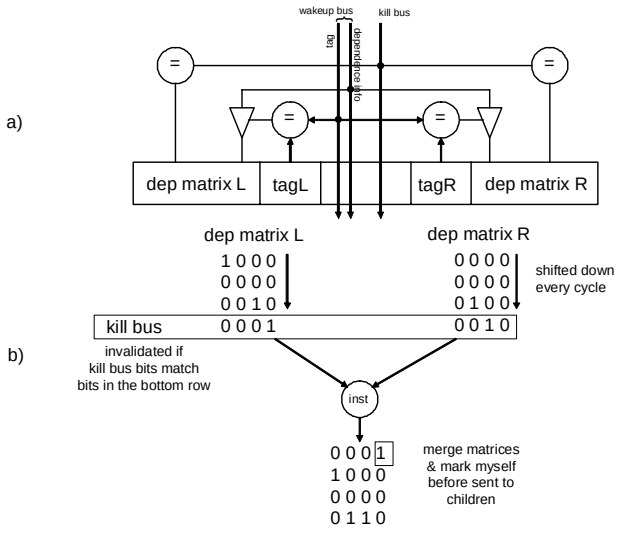


Figure 2. Half-Price Selective Replay Mechanism. (Figure from [9])

matrix shifts down one row, to keep the information consistent with the movement of instructions through the pipeline ahead.

When a load latency misprediction occurs, the execute stage sets the appropriate bit in a *W*-bit wide array called the *kill bus*. These bits are broadcast to every instruction in the scheduler. As is illustrated in part b) of Figure 2, if the kill bus has a 1 in the same column as a 1 in the bottom row of a tag's dependence matrix, that operand's ready bit is reset to zero, as it is dependent on the instruction with the mispredicted latency. In other words, the matching bits indicate that the operand is dependent, either directly or indirectly, on the mis-scheduled instruction.

When an instruction leaves the scheduler window, it merges the dependence matrices that its input operands have received from their parent instructions and marks its own location. It then broadcasts this matrix, along with its destination tag, to the rest of the instructions in the window. (It must also write these bits into a table in the register rename stage for the benefit of dependent instructions that may have not entered the window yet.) When instructions in the window match the input operand on the tag bus, they also latch the dependence matrix of the broadcasting parent instruction. This process propagates dependence information from parent to child during the wakeup phase, giving them knowledge of ancestor instructions further up the dependence tree.

3.1.1 Tag Elimination Compatibility

As is discussed in [9], this selective replay scheme is not compatible with reduced-tag schedulers. Because reduced-tag scheduling makes decisions based on operand availability, problems arise when this availability information is allowed to change after instructions enter the window. Broadcast-based replay relies on every operand in the window tracking its dependencies. Because reduced-tag schedulers gain their complexity benefit by removing some operands from the tag bus, this is not possible.

In schedulers that use non-speculative tag elimination (*i.e.* they do not use last-tag speculation), instructions

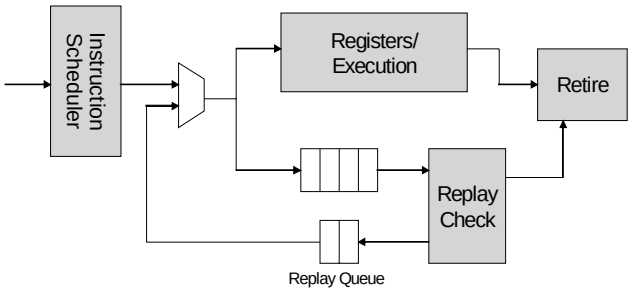


Figure 3. Intel Selective Replay Mechanism.

entering the window would get the proper dependence matrices from the table in the rename stage and they would still be able to monitor the kill bus. However, if the ready bit of an operand that has no comparator needed to be reset, there would be no way for that operand to return to snooping the tag bus.

Furthermore, removing the early arriving operands from the tag bus in last-tag speculation windows makes it impossible for those operands to receive their propagated dependence information from a broadcasting parent instruction.

3.2 Replay using Timed Queues

There are, however, other ways to implement selective replay. Some of these mechanisms differ from the parent/child broadcast model in that, instead of re-executing dependent instructions, they insert a delay into an instruction's execution latency, often through use of a queue or other type of separate instruction storage. The specific mechanism we outline here is derived from a technique proposed in U.S. Patent 6,212,626 [12], held by Intel for inventors Merchant and Sager of the Pentium 4 architecture team [7].[1] A block diagram of this design is shown in Figure 3.

As instructions approach execution, they are also processed through a replay check, which consists primarily of a table of register ready bits. Just before entering the execute stage, instructions look up the status of their input operands in the checker. If the table indicates that the operands are ready, the instruction is allowed to enter execution and retire normally.

If, however, the check table indicates that an operand is unavailable, the instruction sets its output operand as not ready in the table and returns to execution via a replay queue and mux. The replay mechanism informs the scheduler of the presence of an approaching replayed instruction, so that nothing is scheduled into that slot in the same cycle. It is important to note that, in order to maintain forward progress, replaying instructions must always have priority over any work that would be coming out of the instruction scheduler.

When the replayed instruction reaches the input mux, it is sent back into the pipeline as if it had just been issued by the scheduler. On reaching execute, it checks its operands again, just as it did before, to determine whether it needs to replay again (An instruction may have to replay several times to tolerate an L2 cache miss, for example).

In this scheme, the propagation of dependence information is accomplished by the cascading ready bit manipulations in the check table. If there is a latency misprediction, the offending instruction's output will not be set as ready, which will trigger a replay for its children, which in turn will cause a replay for its children's dependents.

It is not specified in the patent exactly how many issue slots coming from the scheduler are stopped when an instruction replays. In our evaluation, we only prohibit the scheduler from issuing into the specific slot that the replaying instruction will be using. This allows the scheduler to issue instructions in the other issue slots.

3.2.1 Tag Elimination Compatibility

A key feature of the Intel replay mechanism is that it maintains the relative timings of instructions throughout the replay sequence. Once a mis-speculated instruction completes, its dependents are replayed just as they were originally scheduled out of the window, only the entire stream has been delayed to accommodate the unexpected extra latency. Consequently, there is no need to "re-schedule" instructions individually, as the previously selected schedule is still valid.

Because replayed instructions are not returned to the scheduler window, there is no extraneous dependence information kept in the window itself. Therefore, reduced-tag schedulers are fully compatible with the Intel-style selective replay.

In schedulers that use last-tag speculation, a last-tag misprediction still results in a one-cycle flush. This recovery is necessary to stop the wakeup of instructions dependent on the last tag misprediction.

4 Replay Evaluation

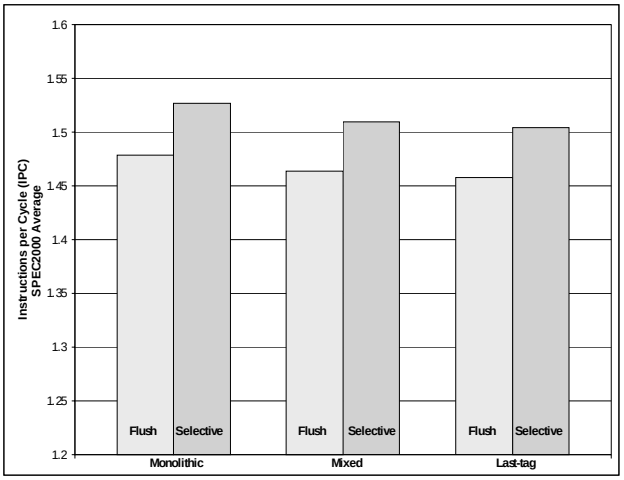
4.1 Simulation Methodology

The architectural simulators used in this study are derived from the SimpleScalar/Alpha version 3.0 tool set [1], a suite of functional and timing simulation tools for the Alpha AXP ISA. The timing simulator executes only user-level instructions, performing a detailed timing simulation of an aggressive dynamically scheduled microprocessor with two levels of instruction and data cache memory. Simulation is execution-driven, including execution down any speculative path until the detection of a fault, TLB miss, or branch misprediction.

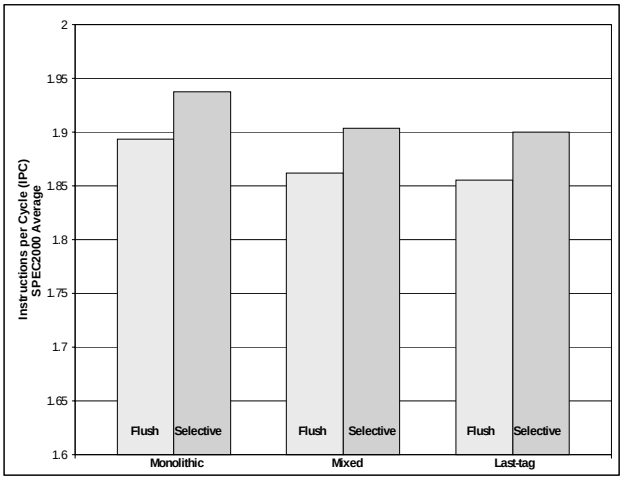
To perform our evaluation, we collected results from all 25 of the SPEC2000 benchmarks [18]. All SPEC programs were compiled for a Compaq Alpha AXP-21264 processor using the Compaq C and Fortran compilers under the OSF/1 V4.0 operating system using full compiler optimization (-O4). The simulations were run for 100 million instructions using the SPEC reference inputs. We used the SimPoint toolset's Early SimPoints [16] to pinpoint program locations to simulate for peak accuracy. Simulation parameters are shown in Table 1.

The simulators were modified to separate reservation stations from the re-order buffer. They were also given support for reduced-tag windows and last-tag prediction. Finally, we were able to simulate either 21264-style flush replay [3] or selective replay using one of the two methods outlined in Section 3.

1. Although we may refer to this as the "Intel" replay mechanism throughout this work, we are making no claim as to whether or not this technique is used in any of their microprocessors, commercial or otherwise. We are only presenting the idea proposed in the publicly available patent documentation.



Issue width 4



Issue width 8

Figure 4. Effect of Selective Replay on Reduced-Tag Schedulers.

Table 1. Simulation Parameters

Parameter	Values
Execution	256-Entry ROB 4- or 8- wide issue 128-Entry Instruction Scheduler Window "Mixed" windows have 1/2/1 ratio "Last-tag" windows have 0/3/1 ratio replay with 4-cycle *load shadow*
Function Units	8 Integer ALU/MULT/DIV, 4 memory ports, 8 FP ALU/MULT/DIV
Branch Prediction	8k entry GSHARE with 8 bits of global history 2K entry BTB, 8-entry RAS
Last Tag Prediction	4k entry GSHARE (only for "Last-tag" configurations) 1-cycle flush misprediction penalty
Memory System	32KB 4-way associative L1 Instruction and Data Caches with 1-cycle latency, 256KB 4-way associative unified L2 with 16-cycle latency, 100 cycle main memory latency across a 16-byte bus

4.2 Tag Elimination and Replay

The SPEC benchmarks were simulated with three different schedulers on both 4- and 8-wide issue configurations, with the results shown in Figure 4. The baseline scheduler ("Monolithic") and the reduced-tag schedulers ("Mixed" and "Last-tag") all gain 2-3% performance improvement due to the decreased replay penalty. *Galgel* benefitted the most with a 26% improvement due to a large number of memory references and enough parallelism to suffer from pipeline flushes. No benchmarks saw a performance degradation due to selective replay. The performance improvement of 2-3% would close much of the gap demonstrated in the experiments of Kim and Lipasti [9].

4.3 Instruction Window Pressure

As is discussed briefly in Borch *et al*.'s work [2], the parent-child broadcast replay model requires that instructions must remain in the scheduler window for several cycles after they are issued, in order to monitor the kill bus for a replay indication. When all of an instruction's ancestors are safely into execution, only then will it finally release its reservation station.

As a result of keeping instructions in the window beyond their issue time, this mechanism can suffer from a reduction in effective scheduler window size. For example, an 8-wide machine with a 4-cycle *load shadow* could be holding as many as 32 instructions in the scheduler window that have already issued, reducing the number of spots that are available for newer instructions. For the typical case, the number of extra instructions held in the window will not be that large because the processor will not usually be filling all of its issue slots.

Using the Intel replay technique, instructions never re-execute out of the scheduler window, thus removing the need to stockpile instructions after they've issued. This reduces the instruction pressure in the window, allowing more work to flow into the empty slots.

On the other hand, the Intel approach can limit execution bandwidth when too many instructions are in replay, thus preventing new instructions from entering execution. However, if there is a large number of replaying instructions, either they are all waiting for one long-latency load, or there are multiple outstanding latency mispeculations. In either case, it is not likely that much more parallelism could be found anyway.

Schedulers with 32, 64, and 128 entries were simulated using both replay techniques, with the results shown in Figure 5. As intuition would suggest, the most benefit was seen in configurations with smaller windows and wider issue, with the 32-entry 8-wide scheduler receiving a 5% performance improvement from the reduced instruction window pressure.

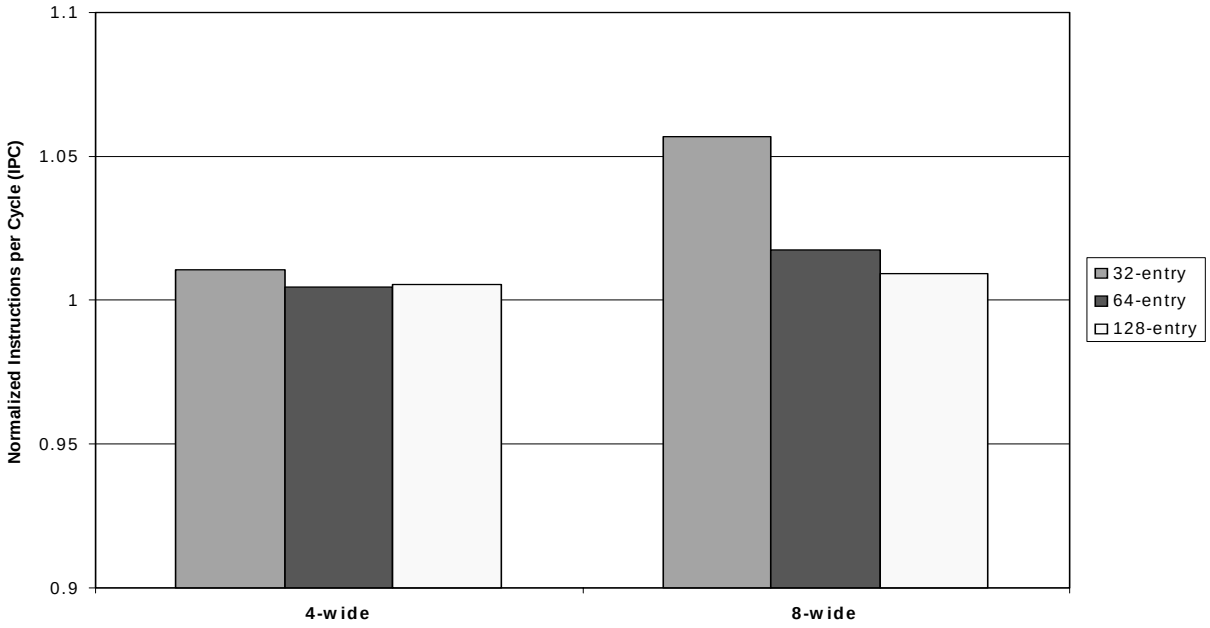


Figure 5. Effect of Reduced Scheduler Pressure. Results presented show the relative performance of a scheduler using the Intel replay mechanism with respect to a scheduler of the same size with a broadcast-based mechanism.

4.4 Power Consumption

In the parent-child broadcast mechanism, the dependence matrix of an issuing operation is sent to all other instructions in the scheduler window. It has been shown in previous studies [4] that these wire-intensive broadcasts can be very costly from a power standpoint.

The load on the broadcast bus as seen by each dependence matrix bit can be estimated as roughly equivalent to that seen by each destination tag bit. While each bit of the matrix bus could be separated from the matrix latches by a low-capacitance pass gate, the bus line must still be able to drive the input of the latch if this gate is open. While having all of the pass gates open is an extreme case (all instructions depend on the broadcast through both operands), it is necessary to take it into account as a peak case. The kill bus bits will also have the same load as the tag bits, since they are also being driven to comparators for each operand.

In a standard instruction scheduler without this replay mechanism, the number of bits broadcast each cycle is

W x (dest tag bits),

where *W* is the scheduler issue width and the number of destination tag bits is $log_2(\#\ of\ physical\ regs)$. If the parent-child broadcast mechanism is incorporated, the number of bits broadcast each cycle is

W x (dest tag bits) + (W x (W x D)) + W,

where *D* is the number of cycles in the load shadow. The first portion of the equation represents the destination tag broadcasts, and it is the same as for the standard window. The second and third terms of the equation represent the dependence matrix bits and the kill bus bits, respectively.

This drastic increase in broadcasts may not directly alter the cycle time (although the layout expansion could have some effect). However, the power consumption will likely be noticeably larger. For example, an 8-wide window with a load shadow of 4 and 256 registers will need to broadcast 328 bits across the scheduler instead of just 64.

The Intel replay technique requires none of these extra broadcasts. The mechanism does include some extra logic, but the power consumed by the check table should be less than the amount that would be dissipated across wire-intensive broadcast lines. This comparison is similar in scope to the comparison of the power usage of a last-tag predictor table with the power used by the scheduler window given in [4].

5 Related Work

Several researchers have recently made the observation that benefit can be gained from removing long latency instructions from the scheduler window as soon as possible. LeBeck's WIB scheduler identifies instructions dependent on long latency operations (data cache misses), and directs these operations to a secondary scheduler [10]. When the long latency operation nears completion, the dependent operations are dumped *en masse* into a small CAM-based dynamic scheduling window. Morancho used a similar approach to move dependent operations following long latency instructions out of the instruction window [13]. Unlike the WIB, they record relative instruction latencies to simplify the re-execution of operations once a valid schedule has been built. The Intel mechanism utilizes a similar approach. As instructions replay, dependencies between dependent operations are maintained by their spacing in the scheduler queues. A schedule is picked and fully committed to for the lifetime of the instruction. Our recent work on the Cyclone scheduler [5] takes this method a step further and replaces the scheduler window

with a dataflow pre-scheduler and timed queues. A queue-based replay mechanism is relied upon to accommodate any incorrectly scheduled instructions.

A number of previous efforts have utilized the register forwarding infrastructure to initiate selective instruction re-execution. The sentinel scheduling technique [11] used "poison bits" contained in the register file that were set when load instructions faulted or did not complete. A branch back to the start of the faulting code would then selectively re-execute the faulting code sequence. As instructions read their registers, only those instructions with poison operands needed to re-execute. The approach is quite similar to the Intel replay queue approach, except instead of redirecting program control, instructions themselves are redirected back into the replay queue. Poison bits were employed in a similar manner by Rogers [15].

6 Conclusions

In the interest of minimizing the performance penalties of deep pipelining, modern processors include selective replay mechanisms to reduce the number of instructions lost due to latency mispredictions. Because these designs may also wish to use complexity-reduction techniques such as tag elimination to improve the performance of dynamic scheduling, it is important for the selective replay implementation to be as unintrusive as possible to the instruction window.

The Intel-style selective replay allows for optimizations such as tag elimination by having its mechanism almost completely external to the instruction scheduler structure. In addition, selective replay mechanisms that are queue and table based have the benefits of both less instruction pressure on the issue window and favorable power characteristics.

Acknowledgements

This work was supported the National Science Foundation CADRE program, grant no. EIA-9975286, and by a National Science Foundation CAREER award, grant no. CCR-0093044. Equipment support was provided by Intel Corporation.

References

[1] Todd Austin, Eric Larson, Dan Ernst. SimpleScalar: an Infrastructure for Computer System Modeling, *IEEE Computer*, Volume 35, Issue 2, Feb. 2002.

[2] E. Borch, E. Tune, S. Manne and J. Emer. Loose Loops Sink Chips, In *Proceedings of the 8th International Symposium on High Performance Computer Architecture*, January 2002.

[3] Compaq Computer Corporation. *Alpha 21264 Microprocessor Hardware Reference Manual*, June 2001.

[4] Dan Ernst and Todd Austin. Efficient Dynamic Scheduling through Tag Elimination, In *Proceedings of the 29th International Symposium on Computer Architecture* (ISCA-29), May 2002.

[5] Dan Ernst, Andrew Hamel, and Todd Austin. Cyclone: A Broadcast-free Dynamic Instruction Scheduler with Selective Replay, In *Proceedings of the 30th International Symposium on Computer Architecture* (ISCA-30), June 2003.

[6] A. Hartstein and T. Puzak. The Optimum Pipeline Depth for a Microprocessor, In *Proceedings of the 29th International Symposium on Computer Architecture* (ISCA-29), May 2002.

[7] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Doug Carmean, Alan Kyker, Patrice Roussel. The Microarchitecture of the Pentium 4 Processor, In *Intel Technology Journal*, 1st quarter 2001.

[8] M. Hrishikesh, N. Jouppi, K. Farkas, D. Burger, S. Keckler, and P. Shivakumar. The Optimal Logic Depth per Pipeline Stage is 6 to 8 FO4 Inverter Delays, In *Proceedings of the 29th International Symposium on Computer Architecture* (ISCA-29), May 2002.

[9] Ilhyun Kim and Mikko Lipasti. Half-Price Architecture, In *Proceedings of the 30th International Symposium on Computer Architecture* (ISCA-30), June 2003.

[10] Alvin R. Lebeck, Jinson Koppanalil, Tong Li, Jaidev Patwardhan, and Eric Rotenberg. A Large, Fast Instruction Window for Tolerating Cache Misses, In *Proceedings of the International Symposium on Computer Architecture* (ISCA-29), May 2002.

[11] S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, W.M. W. Hwu, B. R. Rau, and M. S. Schlansker. Sentinel scheduling for VLIW and superscalar processors, *ACM Transactions on Computer Systems*, 11(4):376--408, 1993.

[12] United States Patent #6,212,626, "Computer processor having a checker", Amit A. Merchant and David J. Sager, assigned to Intel Corporation, issued April 3, 2001.

[13] E. Morancho, J.M. Llaberia, A. Olive. Recovery Mechanism for Latency Misprediction, In *Proceedings of the 2001 International Symposium on Parallel Architectures and Compilation Techniques* (PACT-2001), September 2001.

[14] S. Palacharla, N. P. Jouppi, and J. Smith. Complexity-effective Superscalar Processors, In *Proceedings of the 24th Annual International Symposium on Computer Architecture* (ISCA-24), June 1997.

[15] Anne Rogers and Kai Li, Software Support for Speculative Loads, In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS '92), 1992.

[16] Timothy Sherwood, Erez Perelman, Greg Hamerly and Brad Calder. Automatically Characterizing Large Scale Program Behavior, In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS 2002), October 2002.

[17] Eric Sprangle and Doug Carmean. Increasing Processor Performance by Implementing Deeper Pipelines, In *Proceedings of the 29th International Symposium on Computer Architecture* (ISCA-29), May 2002.

[18] Standard Performance Evaluation Corporation, http://www.specbench.org.

The Impact of Fetch Rate and Reorder Buffer Size on Speculative Pre-Execution

David M. Koppelman
Department of Electrical & Computer Engineering , Louisiana State University
koppel@ece.lsu.edu

Abstract

Pre-execution systems reduce the impact of cache misses and branch mispredictions by forking a slice, a code fragment derived from the program, in advance of frequently mispredicted branches and frequently missing loads in order to either resolve the branch or prefetch the load. Because unnecessary instructions are omitted the slice reaches the branch or load before the main thread does, for loads this time margin can reduce or even eliminate cache miss delay.

Published results have shown significant improvements for some benchmarks, on the order of 20%, with many showing at least single-digit improvements. These studies left unexamined two system parameters that one would expect pre-execution to be sensitive to: fetch rate and reorder buffer size. Higher fetch rate would allow the main thread to reach the troublesome load sooner, but would not affect the slice and so the slice's margin is reduced. Studies have shown large potential margins for slices, but the fetch rate effect has not been measured. A second system parameter is reorder buffer size. A larger reorder buffer would allow a system to hide more of the miss latency that pre-execution reduces.

To test the sensitivity to these factors pre-execution schemes were simulated on systems with varying fetch rates and reorder buffer sizes. Results show that higher fetch rate does not reduce pre-execution speedup in most benchmarks. Reorder buffer size sensitivity varies, some benchmarks are insensitive to reorder buffer size increases beyond 256 entries, but still benefit from pre-execution, the benefit due in large part to prefetching those loads that provide values for frequently mispredicted branches. The benchmarks that are sensitive to reorder buffer size are also the ones that benefit most from pre-execution.

1. Introduction

Pre-execution schemes are one approach to reducing the impact of cache misses and branch mispredictions. In a pre-execution scheme troublesome (frequently missing) loads are identified and for each troublesome load a *slice* consisting of the load and instructions producing its address is constructed and cached. For each slice a *trigger* instruction is identified, the next time the trigger is encountered the slice will be retrieved and executed. If successful the load in the slice, called the *prefetch load*, will execute several cycles before the load for which the slice was constructed, reducing or eliminating cache miss latency. Slices do not affect state visible to the running program and so they may be killed at any time, only at the cost of missing a prefetch opportunity. Slices can also be constructed for troublesome branches, though with the added complication of matching the predicted outcome to the main-thread instruction stream.

In some schemes slices are constructed dynamically and in hardware, in others they are prepared in software based on a profile run. The slice might be a subset of the dynamic instruction stream, or it might be optimized in some way. Slice decode and execution share existing processor resources in some schemes, in others slice execution uses its own decode hardware, execution hardware, or both. For more details see Section 2.

What is common in the pre-execution schemes examined here is that the slice consists of ordinary machine instructions which make use of processor state from the running program, in particular register values, and that the slice is forked when the program reaches some instruction.

The use of machine state and execution resources for slice execution is costly in one way or another. If execution resources are duplicated the cost is direct, and in all schemes at least the register map or register values must be copied. If resources are shared then instruction queues and other scheduling hardware must

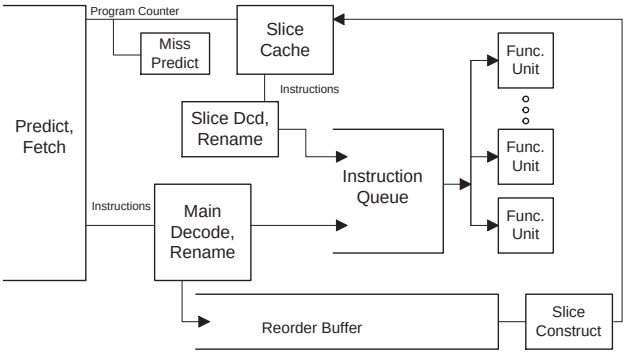


Figure 1. Typical pre-execution hardware organization. Slice instructions share functional units but not reorder buffer slots. Illustrated (and simulated) system has separate decode/rename for slices. In other proposed schemes slice shared decode/rename with the main thread.

be enlarged. When implemented on top of simultaneous multithreading hardware, slices displace ordinary threads.

These costs are offset by the potential benefit, which there is plenty of since load latency is no small problem. Zilles and Sohi [21] show that performance would improve by a large amount, more than doubling for some SPEC2000 integer benchmarks, if troublesome loads hit the cache and troublesome branches were correctly predicted. Pre-execution can prefetch loads which conventional hardware prefetch schemes cannot, such as those generating irregular address patterns.

The various pre-execution schemes described in the literature realize respectable, in some cases large, speedups. For example, Collins, Tullsen, Wang, and Shen demonstrate an average speedup of over 1.3 on memory-intensive benchmarks.

The published analyses of pre-execution schemes looked at factors such as slice construction techniques, but for the most part evaluated pre-execution on a single type of system [1,3,10,11,14,21]. There are two system parameters to which pre-execution may be sensitive: the fetch rate and reorder buffer size. A higher fetch rate will reduce the time advantage of a given slice, or require triggering a slice further back, risking triggering a slice for the wrong path. Reorder buffer size is an important factor because a larger reorder buffer can hide the load miss latency that pre-execution reduces. (Put another way, pre-execution can reduce the latency that would otherwise require a larger reorder buffer to hide.) A related advantage of smaller reorder buffers for pre-execution is that they are more frequently full. That, of course, stops the main thread but not a slice that has been triggered.

The impact of fetch rate and reorder buffer size on pre-execution schemes will be examined here. A pre-execution scheme will be simulated on systems having varying fetch rate, reorder buffer sizes, and slice construction window sizes. The impacts on performance, and the reason for that impact will be analyzed in detail.

The remainder of this paper is organized as follows. A discussion of some existing pre-execution schemes appears in the next section. Pre-execution performance factors are discussed in Section 3. Details of the simulated system and benchmarks are described in Section 4. Experiments are described and discussed in Section 5, related work is discussed in Section 6, and conclusions appear in Section 7.

2. Pre-Execution Schemes

Perhaps triggered by Zilles and Sohi's 2000 study [20] a number of pre-execution schemes had been published in 2001. These will be discussed here, while antecedents and other related schemes are discussed Section 6. An outline of pre-execution and related terminology is presented below, followed by details from published studies and the version simulated here.

2.1. Basic Pre-Execution and Terminology

In a pre-execution scheme *troublesome* loads are identified, these are loads which miss the cache and in some variations are believed to be on a critical path. The instructions preceding a load needed to compute its address are said to be in its *dataflow tree*; the number of preceding instructions considered is called the *construction window size*. When a troublesome load is identified its dataflow tree is constructed, and in some variations optimized; the result is called a *slice*. The slice is placed in a *slice cache*. With a slice constructed for it, the troublesome load is known as a *target* and the corresponding instruction in the slice is called the *prefetch load* (briefly, prefetch). A *trigger instruction* (briefly, trigger) is selected, often the earliest instruction in the construction window. The next time fetch (or more often, a decode step) reaches the trigger the slice is retrieved and *forked*, that is, its execution is started, and will proceed in parallel with the *main thread*. Depending on variation the forked slice may share decode/rename resources and execution resources. The prefetch does not wait if it misses the cache.

2.2. Published Pre-Execution Schemes

Some of the schemes discussed below use pre-execution for branches as well as loads; only the portions handling loads are discussed.

Finding troublesome loads may be harder than it sounds, at least for a run-time mechanism. Only two

schemes identify troublesome loads at runtime, Dynamic Speculative Precomputation (DSP) of Collins, Tullsen, Wang, and Shen [3] and the Slice Processor (SP) of Moshovos, Pnevmatikatos, and Baniasadi [11]. The SP uses a load miss predictor to select target loads (on a miss add 4 to a counter in a PC-indexed table, on a hit subtract 1), the DSP goes further by finding only critical-path loads (based on the time spent waiting at the head of the reorder buffer). The other schemes mentioned here rely on some form of profiling to select target loads.

There are major differences in the way the slice itself is constructed. Unmodified dataflow trees are used by SP, the Dependence Graph Precomputation (DGP) scheme of Annavaram, Patel, and Davidson [1], and speculative Data Driven Multithreading (DDM) of Roth and Sohi [14]. Tree construction occurs dynamically (at run time) in SP and DGP; in SP the tree is constructed from the last 32 (nominally) committed instructions while in DGP it is constructed from instructions waiting in the fetch queue. In DDM tree construction occurs in a pre-processing step.

There are many opportunities to optimize instructions in the dataflow tree, for example replacing or eliminating store/load pairs, and combining arithmetic instructions (say, two instructions incrementing the same variable). The DSP optimizes slices dynamically while Zilles and Sohi [21] in what will be called the Speculative Slice Study (SSS) here, construct their slices by hand.

A refined approach to slice selection and construction, called the Quantitative Framework (QF) here, is presented by Roth and Sohi [15]. A score is constructed for a candidate slice using the margin of the prefetch over the target load adjusted for the number of times the target load will be reached, the score also includes the number of instructions in the slice (which reduces the score). Slices are selected from a set of candidates based on how well they cover target loads, combining the effect of a short, low-margin slice against a longer, high-margin slice that is less likely to reach the target.

In some cases the slices can contain loops. When constructing slices DSP hardware specifically looks for a second instance of the troublesome load so that the constructed slice can have a loop. A loop counter and call level monitor terminate slices that may loop too long. The hand-constructed slices in the SSS also contain loops and use a maximum iteration counter.

The various pre-execution schemes differ in the resources they use to execute slices. In the boldest of these schemes slices execute as a special thread on a simultaneous multithreaded (SMT) [8] machine, competing for decode/rename and execution resources with the main thread. With this resource sharing there is significant potential for slowing down the main thread. SMT (or SMT-like) execution is done by the DSP, DDMT, and SSS. DSP and DDMT do not compete for reorder buffer (ROB) slots with the main thread; it is not clear whether this is true in SSS. An alternative is to provide separate decode/rename resources for slices, this is done in the SP. The DGP is something of a special case since it operates on instructions in the normal fetch stream, relying on run-ahead of the fetch unit.

Of all the schemes only the DGP provides its own execution resources, in the others slice instructions compete with the main thread for functional units, in some cases at a lower priority.

2.3. The Expensive Slice Machine

A goal of this study is to analyze the impact of fetch rate and ROB size on pre-execution performance, not to find a good cost/performance balance. Therefore a pre-execution scheme was chosen for analysis that gives good performance with little regard for resources used. That is, the size of the slice cache and the number of functional units is large, so as to bring out as much potential performance as possible; it will be called the Expensive Slice Machine (ESM).

Slice construction is performed as loads commit; it is performed instantaneously and for all committing loads generating a *lead miss*. ESM constructs (but does not always cache) slices for every load instruction that generates a lead miss. A lead miss is the miss to the level 1 cache that initiates a cache fill. (A following instruction that access the same address before the data arrives is not said to generate a lead miss.) The slice is constructed by extracting the dataflow tree for the troublesome load using a buffer holding recently committed instructions. The *margin* for a newly constructed slice is computed, the number of cycles by which the prefetch load will precede the target load assuming that when the slice is forked fetch proceeds at the same rate as when it was constructed. If the margin is below 2 cycles the slice is not cached.

A *miss distance* is stored with each slice, this is used to determine whether to fork the slice. When the slice is constructed the miss distance is set to zero, it is also set to zero if the slice's prefetch instruction generates a lead miss; it is incremented if the prefetch does not generate a lead miss. (Any practical system would have a separate load miss predictor.)

The number of simultaneously executing slices is limited to eight. If less than eight are executing the slice cache is probed using the addresses of decoding instructions. If a slice is found and its miss distance is less than 7, execution forks. The slice is decoded

and renamed at a rate of 4 instructions per cycle in the base machine. Slice instruction execution proceeds normally, that is, decode, rename, and scheduling proceed at the same speed as for ordinary instructions.

Slice instructions do not use ROB slots but they share execution resources of main-thread instructions. In the systems analyzed there are plenty of functional units and so there is rarely any waiting.

3. Pre-Execution Performance

The speedup attained by pre-execution depends in part on the margin of the prefetch over the target and by the number of targets that lie on a critical path.

3.1. Critical Path and Sensitivity to ROB Size

In a dynamically scheduled system some load misses do not impact execution time, instead the processor "catches up" when these loads complete. Those that do impact execution time in a particular processor configuration are called *critical loads* for that configuration.

There are two ways loads can be critical: they can delay branch or jump target resolution, or they can contribute to the filling of the reorder buffer. The first type will be called *control-critical loads*, the second type will be called *window-critical loads*. Control-critical loads are those loads that, because they don't hit the cache, force instructions computing a branch condition or target for a mispredicted branch to wait. For window-critical loads if the load had not missed the reorder buffer would have filled later, if at all. In both cases the load delays correct-path instruction fetch.

Pre-execution will reduce the miss delay of both types of critical loads, the data presented below shows these effects separately. Reducing the impact of control-critical loads is something that is difficult to do by means other than larger or faster caches or better branch predictors. In contrast, the impact of some (perhaps most) window-critical loads can be reduced by increasing the size of the reorder buffer. For that reason, reorder buffer size sensitivity is important, and is being investigated here.

Apart from just pre-fetching loads early, another way that pre-execution, at least the published schemes, improves performance is by allowing a slice to be fetched and executed when the rest of the system is stalled due to a full reorder buffer. This increases the margin when the target load is caught waiting outside a full reorder buffer while the pre-fetch executes. (In most of the previous studies slices do not share reorder buffer slots with the main thread.)

3.2. Sensitivity to Fetch Rate

As used here the *fetch rate* is the number of decoded instructions (including those that will be squashed) divided by the number of cycles at which decode was not stalled (due to a full reorder buffer, lack of physical registers, etc). The fetch rate is determined by decode width and by the front end, the part of the processor that predicts and fetches instructions. A processor with a decode width of 8, that is, an 8-way superscalar processor, has an ideal fetch rate of 8 instructions per cycle. Due to limitations of the instruction cache and the ability of the front end to predict multiple control transfers per cycle, the front end may deliver less than eight instructions per cycle. Advanced front ends, such as trace caches [13] and multiple branch predictors [19] have higher fetch rates, approaching the maximum possible (the decode width).

Assuming no stalls around the time a slice is forked, the margin for the slice is determined by the fetch rate, the trigger-to-target distance, and the dataflow distance. That there is plenty of margin for troublesome loads was revealed in the study of Zilles and Sohi [20]. They identify troublesome loads, in particular those that have the most impact on overall execution, and plot the number of instructions in the dataflow tree versus the distance from the target load. They show that from 5% to 30% or more of instructions preceding a load are in its dataflow tree, the lower number assuming all stores can be omitted and the higher including all stores. That this margin could be exploited was demonstrated in the many projects described above.

One important question unanswered by these studies is the degree of sensitivity to fetch rate. In most of these studies some estimate is made of margin and slice candidates not meeting this margin are rejected. A higher fetch rate would mean more slice candidates would be rejected (or would be ineffective), and so reduce the impact on performance. Based on observed distribution of instructions [20] moving the trigger back would increase the margin, but at the risk of placing the trigger before a frequently mispredicted branch. Only the QF takes this slice survivability into account [15].

4. Evaluation

4.1. Simulator

The systems were analyzed using RSIM [12], a detailed microarchitecture simulator. Modifications were made to simulate pre-execution and many other unrelated modifications were made. RSIM is a microarchitecture simulator which simulates a dynamically scheduled superscalar processor and memory system. The processor implements a subset of the SPARC V8 ISA [18]. Benchmark programs are compiled exactly as they are for a real system. Linking is identical except for

Table 1. Benchmarks. (Table numbers in millions.)

Bench-mark	Insn Skip'd	Insn Sim'ed	Loads Cmtd	L2 Hits	L2 Misses
mst	1200	500	78	2.34	18.88
em3d	120	300	35	0.33	0.40
bzip2	0	305	78	9.61	3.00
gcc	0	607	108	5.21	0.26
gzip	0	636	108	26.53	1.86
mcf	1000	100	27	2.07	12.83
perl	0	181	35	2.59	0.07
swim	0	400	104	9.85	44.11
TEX	0	102	20	0.58	0.03
vpr	5000	300	69	4.74	3.87
wupwise	5000	500	170	0.28	0.95

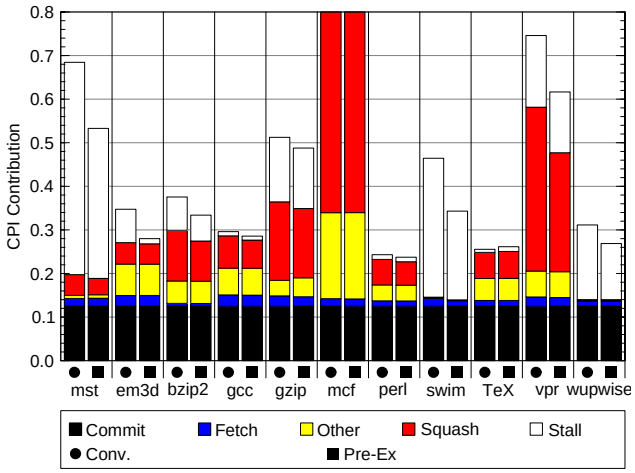


Figure 2. Performance of a conventional system and one using pre-execution. Segments show CPI contribution.

the use of static libraries (though still the system's libraries, not specially prepared versions) and a special startup file. System calls are not simulated.

Dynamic execution is aggressive: The register map used for renaming is checkpointed when branches or jumps are decoded so that recovery can start when mispredicted instructions resolve. Exception recovery is initiated when the faulting instruction is ready to commit. Other system characteristics are summarized below:

Front End

Branch target prediction using a basic block predictor [19] with a 2^{15}-node block address cache; directions predicted with a variation on a YAGS predictor[4] using a 16-bit GHR. Base system has three instruction cache ports and predicts two blocks per cycle. One- and two-icache-port systems predict one block per cycle; four-port system predicts three blocks per cycle. Block predictions are queued. Indirect jump predictor uses a 2^{16} entry GHR-indexed table. Returns predicted with an 8-entry RAS.

Core

Base system 8-way superscalar; three cycle delay from decode to earliest execution opportunity. Reorder buffer holds 256 entries; virtually unlimited functional units. Instruction queues and load/store queues have virtually unlimited space.

Memory

L1 instruction cache: 328 kB, 5-way. L1 data cache: 16 kiB, 4-way, 64-byte line; 2-cycle hit latency including address generation. L2 data cache: 256 kiB, 8-way, 16-cycle hit latency. Memory, 100 cycle access latency plus congestion and overhead.

Pre-Execution

Slices constructed from 256-instruction window; maximum slice size 64 instructions; slice rejected if margin (prefetch to target) less than 2 cycles. Slice cache size 2^{16} entries. Slice not forked if target load has more than 6 consecutive non lead misses (a hit or miss to a line already on the way). At most 8 slices in flight; slices injected at half the decode width (4 instructions per cycle in base system). Slices execute with shared execution resources, but use private decode and rename and do not use reorder buffer slots.

4.2. Benchmark Programs

The simulated programs come from the SPEC and Olden suites. Except for vpr, the SPEC programs were not selected for pre-execution suitability. (That is, no program was selected because of favorable speedup or any other performance reason.) The Olden benchmarks are a set of pointer-intensive microbenchmarks adapted by Luk [9] for use in studying prefetching and used later by other investigators to test the effectiveness of pre-execution and other schemes. They are included here so results can be compared to other studies that use these benchmarks.

Benchmarks vpr, swim, gzip, mcf, and wupwise are compiled using the SPEC CPU2000 makefiles, using code from that suite. The code for the other benchmarks was obtained from their standard distributions, compiled with optimization. Optimization was targeted to an UltraSPARC II processor, so scheduling would not perfectly match the wider-issue systems simulated here.

Benchmark swim uses test inputs, mcf and wupwise use reference inputs. Benchmark vpr uses reference inputs but is only run for placement. Olden benchmark em3d uses input 25000 100 75 1, benchmark mst uses input 3407 1. The other spec benchmarks use shortened inputs. Table 1 summarizes benchmark characteristics including the portion of the benchmark simulated.

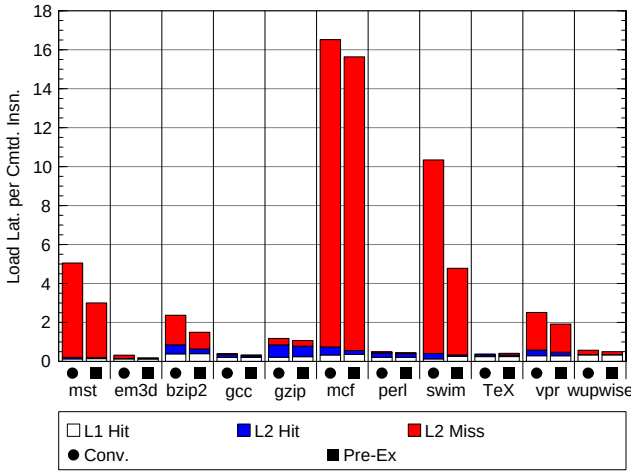


Figure 3. Load latency per committed instruction of a conventional system and one using pre-execution. Segments show contribution from level 1 hits, level 2 hits, and misses.

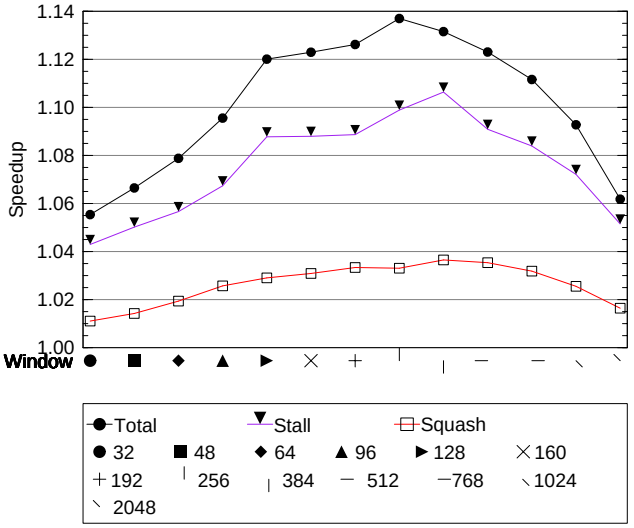


Figure 4. Speedup of pre-execution with varying construction window sizes. Total is the actual speedup, Stall shows the speedup due to window-critical loads, and Squash shows the speedup due to control-critical loads.

5. Experiments

The performance of conventional and pre-execution systems is shown in Figure 2. The total height of each bar is the execution time in cycles per instruction (CPI), the segments show the ideal execution time, **Commit**, and four factors degrading performance: correct-path stalls, **Stall**, misprediction squashes and wrong-path stalls **Squash**, instruction cache port limitations **Fetch**, and miscellaneous factors, **Other**. Control-critical loads primarily effect the **Squash** segment while window-critical loads primarily effect the **Stall** segment.

The size of each segment is determined by tallying how each decode slot is used at each cycle and dividing each tally by the number of committed instructions times the decode width. The **Commit** segment shows slots used by committed instructions, its height is the ideal execution time of $\frac{1}{8}$ CPI. The **Fetch** segment shows decode slots unused due to a limit on the number of instruction cache ports. The **Squash** segment shows slots wasted due to mispredicted or unpredicted control transfers. This includes slots holding instructions that survive long enough to be scheduled but are ultimately squashed due to mispredictions and slots empty because the ROB is full while waiting to fetch down a mispredicted path. The **Stall** segment shows slots empty because the ROB is full while waiting to fetch down a correct path. The **Other** segment shows other cases. For most benchmarks this is dominated by instructions that are squashed before they could be scheduled and instruction cache misses.

The benchmarks show variation in their CPI, the performance loss due to control- and window-critical loads, and in how much pre-execution helps. Benchmarks perl, TEX, and gcc are the most efficient and are little improved by pre-execution while most of the less efficient benchmarks enjoy more substantial speedup with the exception of gzip, which is ILP limited.

The benchmarks' sensitivity to pre-execution is determined in part to the number of cache misses. Load latency per committed instruction is plotted in Figure 3. The segments show the contribution of loads that hit the level 1 cache, hit the level 2 cache, and those that miss the level 2 cache. Benchmark mcf has the longest load latency but because the loads that miss the cache are participating in a long pointer chase pre-execution can do little to reduce the latency.

5.1. Speedup and Cons. Window Size

Average speedup is plotted in Figure 4 and the speedup of selected benchmarks is plotted in Figure 5. In both plots speedup is shown for pre-execution schemes with construction windows varying from 32 to 2048 instructions. The points marked **Total** show the speedup over the conventional system. Benchmarks vary in the size of the window needed (based on the level 2 cache hit ratio), average performance peaks at a window size of 384. Performance drops with further increase in window size because of the limit of eight in-flight slices. The base construction window size of 256 achieves close to maximum performance.

In addition to total speedup these figures show an estimate of the speedup obtained by improving only control-critical loads, **Squash**, and window-critical loads, **Stall**. The control-critical speedup estimate is

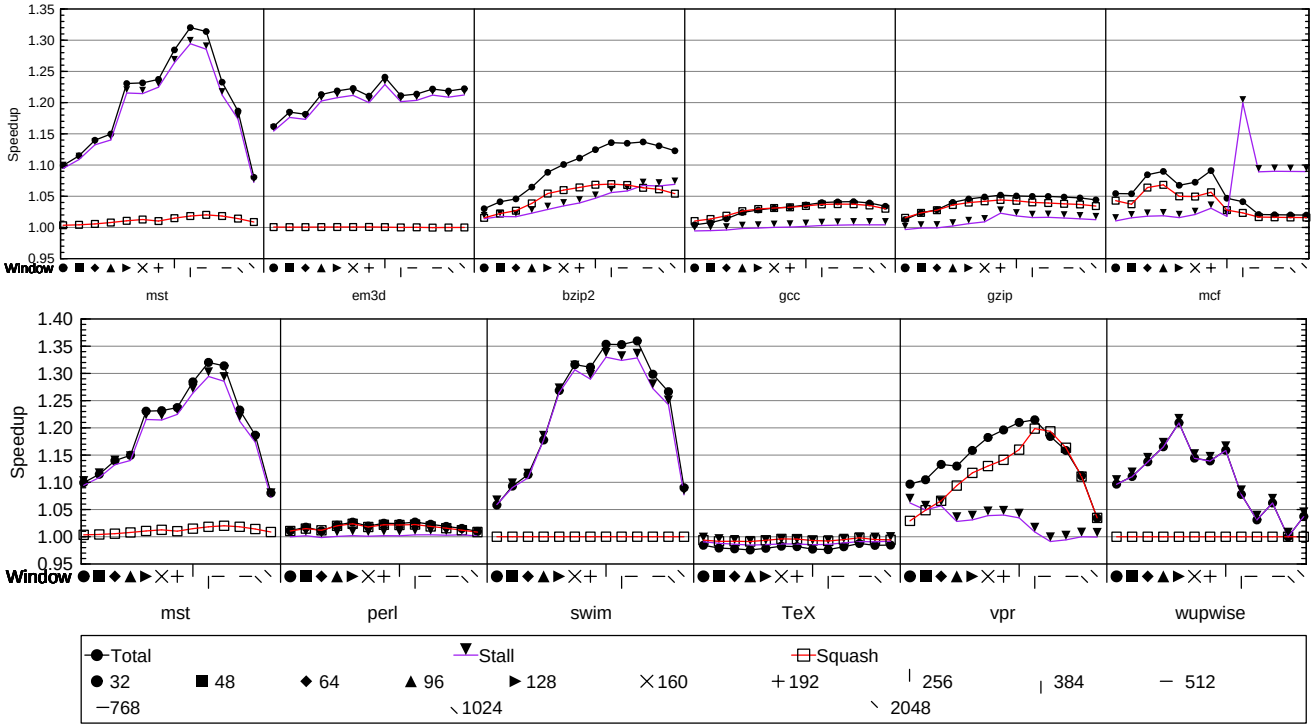


Figure 5. Speedup of pre-execution with varying construction window sizes.

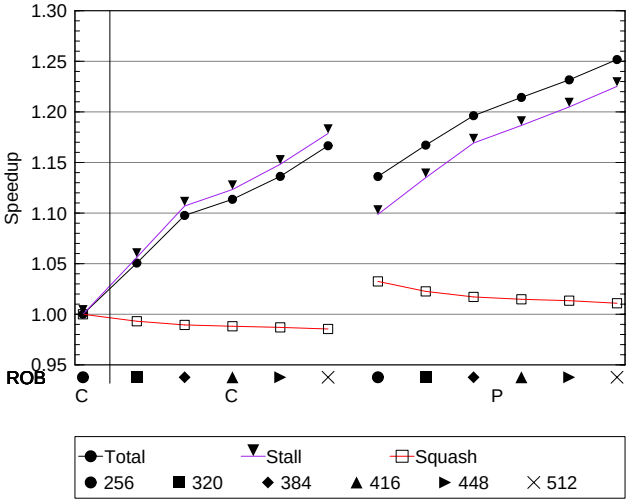


Figure 6. Average speedup of benchmarks on systems with, P, and without, C, pre-execution with varying reorder buffer sizes. Speedup is over conventional system with a 256-entry ROB.

obtained by essentially replacing the **Squash** segments (from Figure 2) of the conventional system with the corresponding segments in the pre-execution system and comparing the result to the unmodified conventional system. A similar approach is used for the window-critical speedup.

From Figure 4 one can see that for these benchmarks most performance improvement is from window-critical loads. Looking at individual benchmarks in Figure 5 one can see that for some benchmarks, such as mst and swim, almost all improvement is for window-critical loads, for others control critical loads are more important.

5.2. Reorder Buffer Size

The impact of reorder buffer size and pre-execution on critical loads is shown in Figure 6 where speedups are plotted for systems with reorder buffer sizes from 256 to 512, the left group is conventional, the right is using pre-execution. All speedups are with respect to the base system (with a 256-entry ROB).

On average the speedup obtained with pre-execution and a 256-entry ROB is about the same as a conventional system with a 448-entry ROB. If both systems are feasible, the less expensive system is better. (Further below larger ROB systems also have longer pipelines.)

As expected, both pre-execution and larger ROBs reduce the impact of window-critical loads. Comparing a 512-entry ROB conventional system to a 256-entry pre-execution system, the two center points, shows better performance on the conventional system. The pre-execution system is not as effective at improving window-critical loads but unlike the conventional system can improve control-critical loads.

Pre-execution is still effective on systems with larger

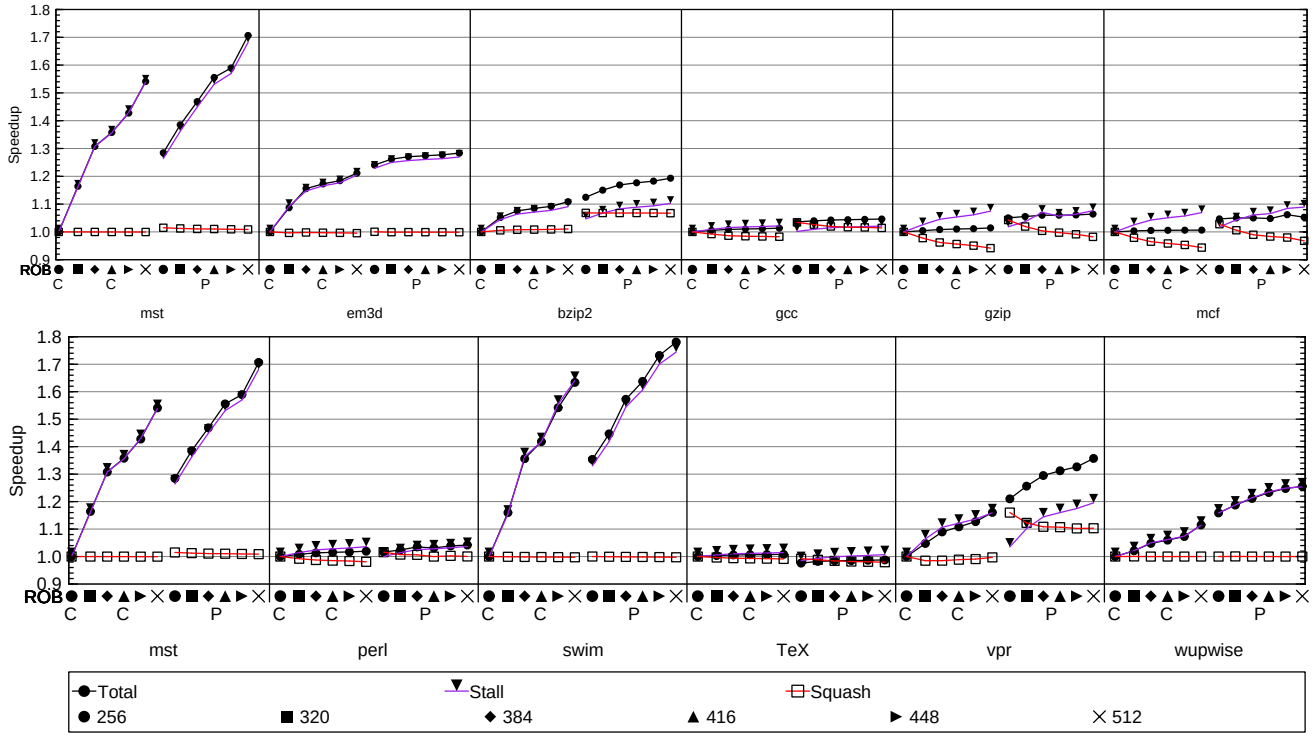


Figure 7. Speedup of selected benchmarks on systems with, P, and without, C, pre-execution with varying reorder buffer sizes. Speedup is over conventional system with a 256-entry ROB.

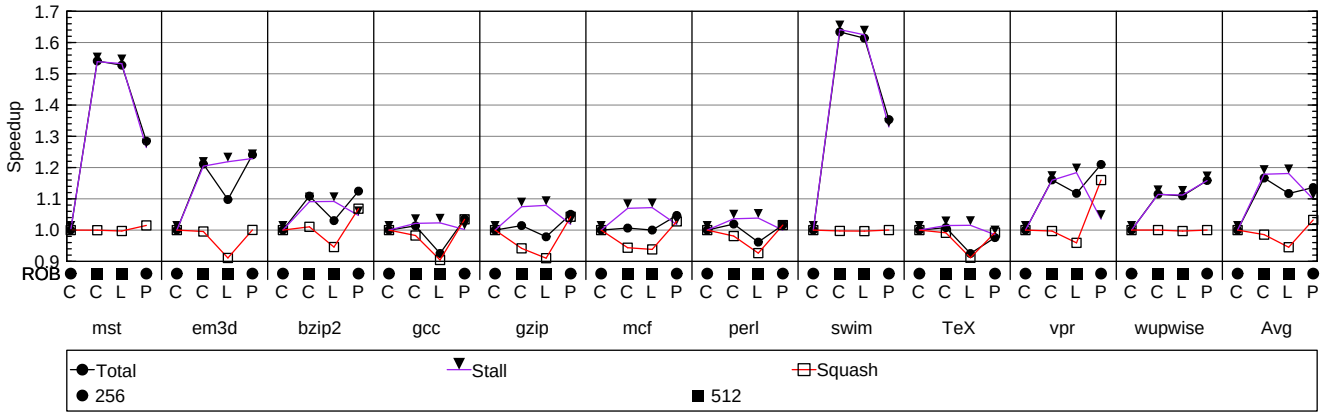


Figure 8. Speedup of a conventional system with a larger ROB, second C, one with a larger ROB but with 3 additional scheduling pipeline stages, L, and a pre-execution system with a 256-entry ROB, P.

ROB sizes, at least up to 512 entries. On the base systems it would take more than 800 entries to cover the level 2 miss latency.

Figure 7 shows the same data for selected benchmarks. Pre-execution is able to out-perform a larger ROB on bzip2, gzip, and vpr due to the control-critical loads that can be helped.

In the comparisons above the systems with a larger reorder buffer got it for "free," in real systems ROB size may be limited by critical paths in scheduling queues needed to hold pending instructions. Figure 8 shows the speedup of three systems: a conventional system with a 512-entry ROB (the same as the one used above), a conventional system with a 512-entry ROB and three additional stages in the scheduling pipeline, **L**, and a pre-execution system using a 256-entry ROB.

The longer scheduling pipeline is felt after branch mispredictions, its impact can be seen in the **Squash** speedup component. For four of the benchmarks this results in a slowdown over the base system, for the others it reduces the speedup over the conventional system with the unmodified pipeline.

When compared to a conventional system with a

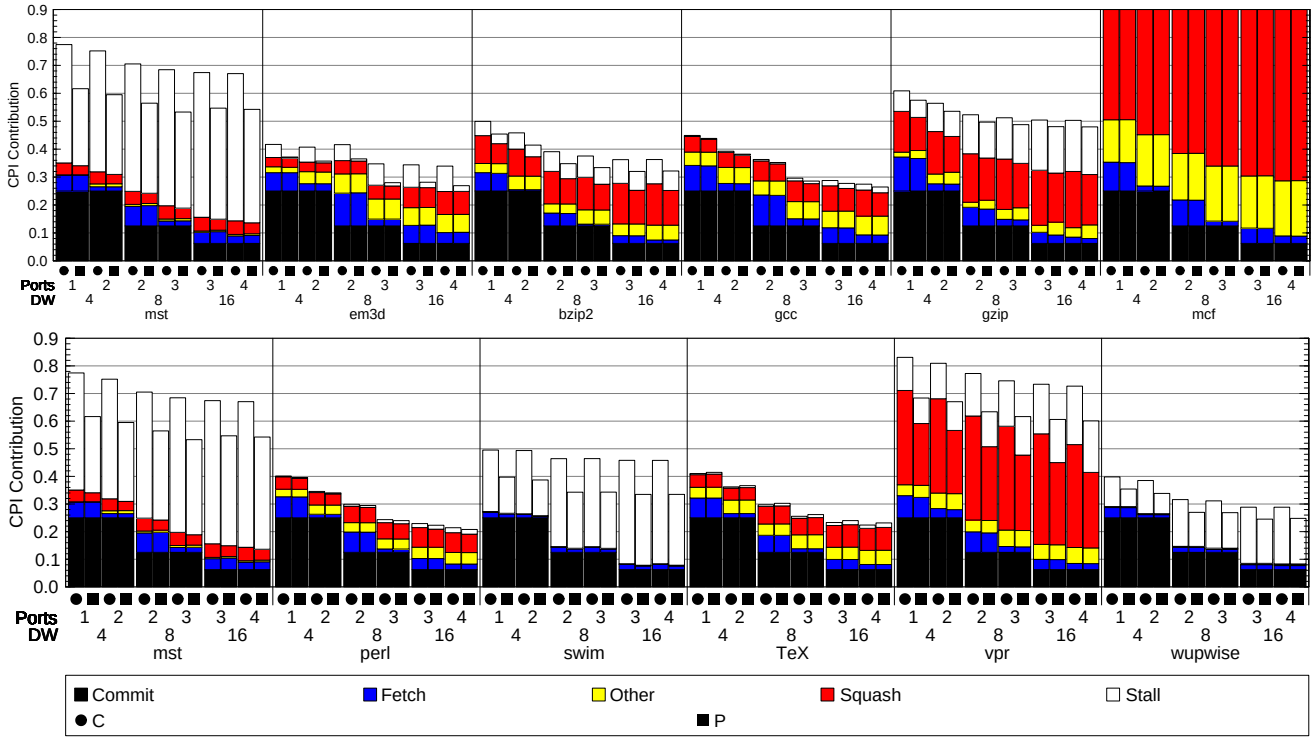


Figure 9. Performance of conventional and pre-execution systems with different front ends.

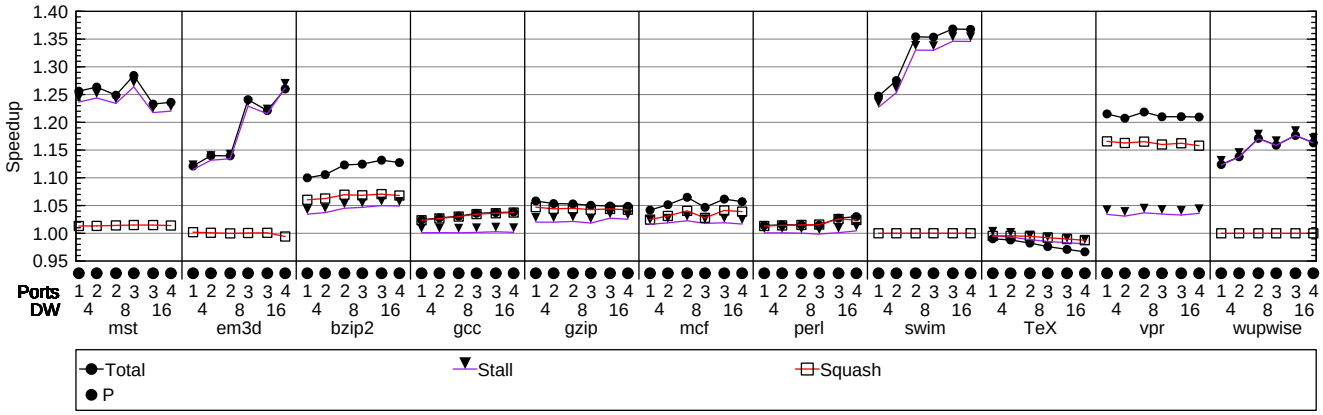


Figure 10. Speedup and speedup contributions of conventional and pre-execution systems with different front ends.

longer pipeline pre-execution does much better, outperforming the conventional system on all but two benchmarks (one of which is synthetic).

5.3. Fetch Rate

Fetch rate is another system parameter that can affect pre-execution. Systems with varying front ends were simulated, including 4-, 8-, and 16-way processors. Front ends that can predict from one to three basic blocks were simulated, the number of instruction cache ports was varied from one to four. The systems with one and two ports predict one block per cycle; the ones with three ports predict two blocks, and the one with four ports predicts three blocks.

The averaged results are plotted in Figure 11 and the results for individual benchmarks are in Figure 9 speedups are shown in Figure 10. In all plots systems are arranged in order of increasing fetch rate The speedup shown is for a pre-execution system compared to a conventional one using the same front end, showing how much added performance can be obtained using pre-execution.

The reduction of pre-execution effectiveness with fetch rate, if present, is small. Several benchmarks show increased speedup with fetch rate as the impact

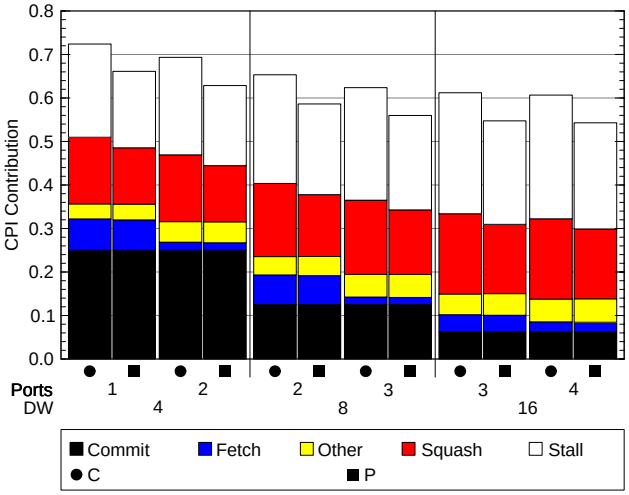


Figure 11. Average performance of conventional and pre-execution systems with different front ends. DW indicates decode width (from 4- to 16-way superscalar), Ports indicates number of instruction cache ports. Segments show CPI contribution.

of control- and window-critical loads becomes a larger fraction of execution time.

6. Related Work

Pre-execution schemes rely on a main thread to repeatedly trigger slices that only run briefly. In contrast the Slipstream Processor of Sundaramoorthy, Purser, and Rotenberg [17] and in systems using Master/Slave Speculative Parallelization of Zilles and Sohi [22] a condensed version of the original program serves the same purpose as a slice: it produces critical values faster than the original program would. Unlike a slice the condensed program is long running and produces more than just troublesome results. Its results are communicated to hardware running the original program where it is used for predictions and later checked for correctness. Correctness needs to be checked because the condensed program, in order to keep ahead of the original, may not always compute correct results.

Pre-execution is just one way of executing a load instruction early. Another is to allow fetch to proceed when some resource limit nears. Instructions fetched under these conditions may prefetch the cache and can provide branch outcomes. Balasubramonian, Dwarkadas, and Albonesi [2] describe such a scheme in which fetch goes on as a *future thread* even when the number of physical registers is low and can discard completed or unneeded instructions, avoiding the reorder buffer size limitation.

Pre-execution schemes are one way of starting loads early, hardware prefetch schemes are another [7]. In such schemes hardware monitors addresses that miss the cache, both level-1 and level-2 prefetchers have been investigated. The hardware, designed to recognize sequential [16], stride [5], or previously encountered patterns [6], generates a prefetch for a predicted next address in the pattern.

Hardware prefetch schemes are quite effective on many memory access patterns, but have trouble predicting many others. Several of the pre-execution schemes have been compared against hardware prefetch and found to complement it well, prefetching addresses that hardware prefetch could not [1,11].

7. Conclusions

Pre-execution improves performance by resolving load addresses and branch directions early using a thread that is forked speculatively upstream of the load. As verified here, pre-execution cannot be outrun by any reasonable front end, faster fetch results in only a small reduction in speedup.

Performance improvement is achieved by prefetching control- and window-critical loads. There are few alternative mechanisms to improve control-critical loads, but the impact of many window-critical loads can be reduced by increasing the reorder buffer size. Data presented here show that for some benchmarks, systems with larger reorder buffers enjoy the same speedup as those with pre-execution, the benchmarks so affected are the ones pre-execution is most effective on. When systems with larger reorder buffers also have longer scheduling delays pre-execution is at a greater advantage. The longer scheduling delays slow many programs while the larger ROB size helps only a few. This is in contrast to pre-execution which (as simulated) only slows one program. One factor not examined here is which is less costly, pre-execution or increased reorder buffer size.

8. Acknowledgments

This work was supported in part by the National Science Foundation under Award No. CCR-0105478 and through an allocation of time on the high-performance computing facilities within LSU's Center for Applied Information Technology and Learning, which is funded through Louisiana legislative appropriations.

9. References

[1] Murali Annavaram, Jignesh M. Patel, and Edwared S. Davidson, "Data prefetching by dependence graph pre-computation," in *Proceedings of the 28th Annual International Symposium on Computer Architecture,* June 2001, pp. 52-61.

[2] Rajeev Balasubramonian, Sandhya Dwarkadas, and David H. Albonesi, "Dynamically allocating processor resources between nearby and distant ILP," in *Proceedings of the 28th Annual International Symposium on Computer Architecture,* June 2001, pp. 26-37.

[3] Jamison D. Collins, Dean M. Tullsen, Hong Wang, and John P. Shen, "Dynamic speculative precomputation," in *International Symposium on Microarchitecture,* December 2001, pp. 306-317.

[4] A. N. Eden and T. Mudge, "The YAGS branch prediction scheme," International Symposium on Microarchitecture, December 1998, pp. 69-77.

[5] J. Fu, J.H. Patel, and B.L. Janssens, "Stride directed prefetching in scalar processors," in *Proc. of the 25th Annual International Symposium on Microarchitecture*, pp. 102-110, 1992

[6] D. Joseph and D. Grunwald, "Prefetching using Markov predictors," in *Proc. of the Int'l Symp. on Computer Arch,* June 1997, pp. 252–263.

[7] Steven P. Vanderwiel and David Lilja, "Data prefetch mechanisms," *ACM Computing Surveys*, vol. 32, no. 2, pp. 174-199, June 2000.

[8] Jack L. Lo, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Rebecca L. Stamm, and Dean M. Tullsen, "Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading," *ACM Transactions on Computer Systems,* vol. 15, no. 3, pp. 322-354, August 1997.

[9] Chi-Keung Luk and Todd C. Mowry, "Compiler-based prefetching for recursive data structures," in the proceedings of *The Seventh International Conference on Architectural Support for Programming Languages and Operating Systems,* October 1996, pp. 222-233.

[10] Chi-Keung Luk, "Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, June 2001, pp. 40-51.

[11] Andreas Moshovos, Dionisios N. Pnevmatikatos, and Amirali Baniasadi, "Slice-processors: an implementation of operation-based predictoin," in *Proceedings of the 15th International Conference on Supercomputing,* 2001, pp.321-334.

[12] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve, "RSIM reference manual version 1.0," Rice University Dept. of Electrical and Computer Engineering, August 1997, Technical Report 9705.

[13] Eric Rotenberg, Steve Bennett, and James E. Smith, "A trace cache microarchitecture and evaluation," *IEEE Transactions on Computers,* vol. 48, no. 2, pp. 111-120, February 1999.

[14] Amir Roth and Gurindar Sohi, "Speculative data-driven multithreading," in *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, 2001.

[15] Amir Roth and Gurindar Sohi, "A quantitative framework for automated pre-execution thread selection," in *Proc. of the 35th Annual International Symposium on Microarchitecture*, pp. 430-441, 2002

[16] A.J. Smith, "Sequential program prefetching in memory hierarchies," *IEEE Computer*, vol. 11, no. 12, pp.7-21, Dec. 1978

[17] Karthik Sundaramoorthy, Zach Purser, and Eric Rotenberg, "Slipstream processors: improving both performance and fault tolerance," in the proceedings of *The Ninth International Conference on Architectural Support for Programming Languages and Operating Systems,* November 2000, pp. 257-267.

[18] David L. Weaver and Tom Germond (eds.), "The SPARC architecture manual, Version 9," Englewood Cliffs, New Jersey: Prentice-Hall, 1994.

[19] Tse-Yu Yeh, Deborah T. Marr, and Yale N. Patt, "Increasing the instruction fetch rate via multiple branch prediction and a branch address cache," in *Proceedings of the International Conference on Supercomputing,* 1993, pp. 67–76.

[20] Craig Zilles and Gurindar Sohi, "Understanding the backward slices of performance degrading instructions," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000, pp. 172–181.

[21] Craig Zilles and Gurindar Sohi, "Execution-based prediction using speculative slices," in *Proceedings of the International Symposium on Computer Architecture,*, June 2001, pp. 2-13.

[22] Craig Zilles and Gurindar Sohi, "Master/slave speculative parallelization," in *Proc. of the 35th Annual International Symposium on Microarchitecture*, pp. 85-96, 2002

A Detailed Study of Hardware Techniques That Dynamically Exploit Frequent Operands to Reduce Power Consumption in Integer Function Units *

Kaushal R. Gandhi and Nihar R. Mahapatra
Department of Computer Science & Engineering
University at Buffalo, The State University of New York
Buffalo, NY 14260-2000, USA
Email: {krgandhi, mahapatr}@cse.buffalo.edu

Abstract *The use of multiple pipelines in superscalar processors and increasing hardware support for intelligent compilers have resulted in processors becoming computationally intensive. There has been an increase in the number of function units, which are wider and operate at higher speeds. A high percentage of integer instructions executed in standard benchmarks reveal a high usage of integer function units.*

In this paper, we analyze three different techniques, namely, memoing (caching results that can be reused), narrow-width operand exploitation (limiting computation to low order bytes), and byte encoding (computation performed over significant bytes), that dynamically exploit operands to lower power consumption in integer function units. Each technique proposes avoiding redundant computation in a function unit and providing the output or a part of the output via alternate means. We analyze and compare these techniques in terms of power savings, area and delay overheads, and their effectiveness when applied to different integer function units. Previously, estimates of power savings based on switching activity were reported. Our implementation of integer function units (CLA, array multiplier, comparator, etc.) at the VLSI layout level and analysis using standard integer benchmarks from the SPEC CPU2000 suite provide realistic power savings, area, and delay overheads.

1 Introduction

VLSI research and development and advances in computer architecture have resulted in processors becoming computationally intensive. The use of multiple pipelines in superscalar processors and increasing hardware support for intelligent and efficient compilers has led to higher performance in processors, but at the same time increased the computation demanded from functional units. The increase in the performance of processors has led to greater power consumption and thrust low power to the forefront as an important design parameter. A high percentage of integer instructions executed in standard benchmarks reveal a high usage of integer function units. Hence the necessity of designing power-efficient integer function units.

Power dissipation in CMOS circuits falls into two categories: dynamic and static. Static power dissipation is due to leakage current. Dynamic power dissipation is due to: (1) short-circuit current during an output transition that is caused by a DC path between the supply rails and the ground line and (2) the charging and dis-charging of capacitive loads during logic changes [9]. Techniques to reduce static power dissipation aim to turn off functional units during idle cycles, e.g., through power gating and use of dual threshold voltages [12, 10, 14, 18]. At the circuit level, voltage scaling, transistor sizing, transformations, and innovative circuit design techniques have been proposed to reduce power consumption [5, 6, 17]. At a higher level of abstraction, compilers aim to reduce power by producing energy-efficient code. Most of these techniques do not exploit operand values dynamically. Only recently, researchers have looked into the possibility of exploiting operand values to reduce power consumption. The observation that many computations on operands are redundant for typical applications and the fact that operands that are inputs to functional units are not completely random provide opportunities for operand exploitation to reduce dynamic power dissipation. Three such techniques are: *memoization* (caching frequent operand sets and results for reuse), *narrow-width operand* (operands that can be represented in fewer bits) *exploitation*, and *byte encoding* (computation skipped over insignificant bytes). In this paper, we comprehensively analyze and compare these techniques in terms of their power savings, area and delay overheads, and effectiveness when applied to different functional units.

The organization of the remainder of the paper is as follows. In Section 2, we describe the above three techniques in detail and in Section 3, we analyze them. Following that, in Section 4, we describe the simulation setup. In Section 5, we analyze the results obtained and compare them. Finally, we conclude and propose future work in Section 6.

*This research was supported by US National Science Foundation Grant # 0102830.

2 Operand value exploitation for low power

We analyze memoization, narrow-width operand exploitation technique, and byte encoding technique [2, 3, 7, 8]. These techniques share some common features: (1) hardware schemes that exploit operands dynamically, (2) detection of frequent operands, (3) reduction of redundant computation by avoiding computation on frequent operands and providing the output via alternate means, and (4) power savings achieved by reduction in the switching activity of a function unit. Memoization exploits the temporal locality of operand sets [2], while the other two schemes exploit the magnitude of operand values at different levels of granularity [3, 7, 8].

2.1 Memoization

Reusing results that were previously calculated, instead of computing them again, could potentially lead to significant power savings in function units. The frequent reuse of data in many applications supports this claim. *Memoization*, a technique that stores results for reuse, has been used previously for improving performance [13]. Azam et al. use this concept to reduce power dynamically [2]. They proposed the use of an execution cache per function unit that can be used as a buffer to capture the temporal locality of operand sets in instruction and data streams. Each entry of the execution cache consists of a computed value and the operand set for which it was computed. The execution cache is either a direct-mapped cache or an s-way set associative cache with LRU replacement policy. An execution cache stage is introduced in the processor pipeline, which is placed between the instruction decode stage and the execute stage, to determine whether an operand set is being reused.

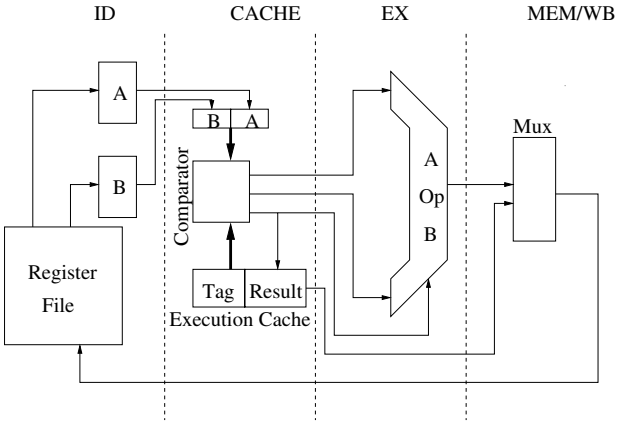


Figure 1: Memoization: Simple bypass via operand tagged caching [2].

During the execution cache stage, certain bits of an operand set ready to be issued to a function unit are used to index into the execution cache. The remaining bits of the operand set are compared to the corresponding bits of the operand set stored in the cache entry. The matching of these two operand sets implies that computation by the function unit can be bypassed as the result is already known and can be reused. If the operand sets do not match, the function unit performs computation over these operands and the result is stored in the execution cache. Using this technique, operand sets are exploited based on their their frequent occurrence for a particular function. The design is illustrated in Fig. 1 [2].

2.2 Narrow-width operand exploitation

Although modern microprocessors support 64-bit addresses and operations, most ALU operands have much smaller magnitudes that can be represented using 16, 8, or fewer bits. By dynamically shutting off that part of a function unit which operates over the high order bytes of frequent operands and sign extension of the result computed for the low order bytes results in significant reduction of switching activity. This approach to operate only a part of the original circuit by latching of certain inputs is termed *guarded evaluation* [16]. We study two implementations of this technique [3, 7].

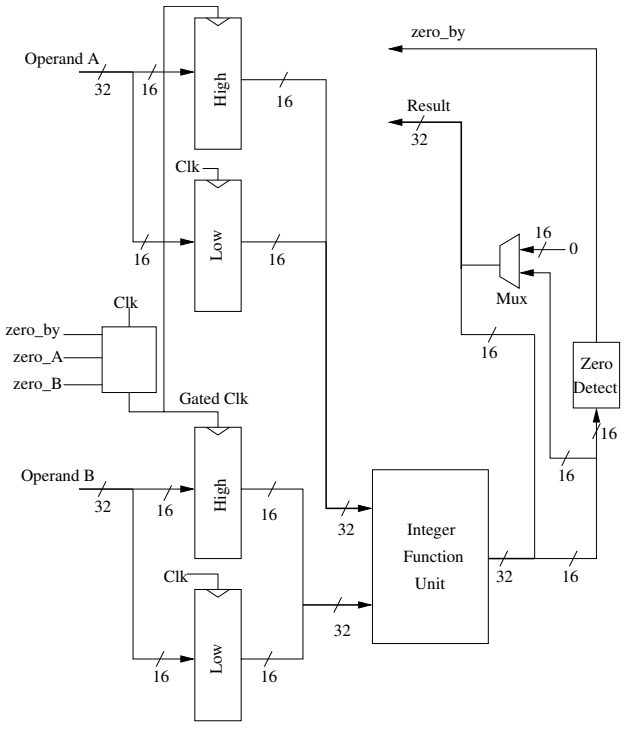


Figure 2: Exploiting narrow-width operands [3].

Fig. 2 illustrates the architecture proposed by Brooks et al. [3]. By disabling value changes in the high order part of a function unit when both operands are narrow-width operands, they are able to reduce switching activity and thus reduce the dynamic power dissipation. This disabling of the high order part of a function unit is achieved by selectively clocking the latches that feed the high order bits of the operands to the function unit. The low order bits are always latched. The selective latching of the high order part of the operand depends upon whether both the operands have leading ones/zeros in their high order bits. The high order bits are latched selectively based on the clock and a signal called *zero/one*. This signal, generated using a *zero/one* detection logic, is stored along with an operand in the register when the operand was either computed as a result or when it was loaded from the cache. The result for the high order part of the function unit is supplied by hardwired values through a multiplexor.

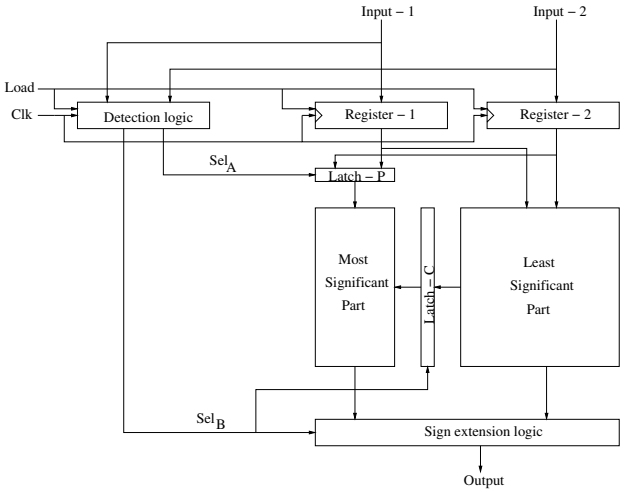


Figure 3: Exploiting narrow-width operands by partially guarded computation [7].

Fig. 3 illustrates the technique employed by Choi et al. to reduce dynamic power dissipation [7]. The high order bits of the inputs are selectively passed on to that part of the function unit that operates on them through latches. The latches are controlled by a signal produced by a detection logic which detects whether the high order bits of both the operands are either zeros/ones. The high order bits of the result are computed via a sign extension logic when both the operands are narrow-width operands.

2.3 Byte encoding

This scheme employs the same method of preventing the propagation of operand bits as described in the previous section, however, at a finer granularity [8]. Canal et al. propose exploitation of operands at various stages in a processor pipeline. We only discuss the application to integer function units here. Operands are coded at a byte level and this choice of dividing an operand into bytes is arbitrary. The least significant byte is always considered to be significant and a function unit always operates over this byte of the operand sets. One scheme of encoding is to add extra bits that represent the number of high order bytes that are merely sign extensions. Another scheme of encoding is to represent each high order byte by one bit. A byte is encoded as 1 if it is the sign extension of the adjacent byte and 0, otherwise. The overhead in terms of bits appended to each operand is higher in the latter scheme, but it exploits more cases of frequent valued operands than the former. The extension bits are computed using simple detection logic when an operand is loaded from the instruction or data cache and when a result is computed. The authors describe three different implementations for a 32-bit wide processor, namely, byte-serial implementation, in which the data path is one-byte wide, semi-parallel implementation, in which the data path is two-bytes wide, and fully parallel implementation, in which the data path is four-bytes wide.

3 Analysis of techniques

In this Section, we analyze the advantages of each scheme and also discuss some of the shortcomings. Each of these schemes involves detection of frequent operand values, a method to bypass or limit computation on such frequent-valued operands, and a method to provide the whole output or a part of it by alternate means.

3.1 Frequent-valued operand detection

Integer function units that have long latencies and cause extensive switching are targets for memoization. The power savings would be significant in such function units if they were to operate on the same operand set frequently. All operand sets that have temporal locality can potentially be exploited. Frequent valued operands as exploited by this scheme are operand sets that have temporal locality with respect to a particular function unit. The implementation by Azam et al. uses an execution cache per function unit [2]. Hence the area overhead is significant. In order to detect the temporal locality of an operand set, an operand set ready to be issued to a function unit is compared to an operand set stored in an entry of the execution cache. This involves a comparison each time an integer function unit is supposed to be used. This is an expensive overhead in terms of the power dissipated

while comparing operand sets. Thus a limitation of this technique is the kind of function that can be bypassed to achieve significant power savings. As the detection of frequent operands involves comparison of operands, it is most useful to apply this technique to functions that consume more power than addition, as comparison of operands, during the cache stage, incurs power consumption similar to that of addition.

Narrow-width operand exploitation targets all types of integer function units in the processor pipeline and results presented previously show significant power savings [3]. Their implementation exploits those narrow-width operands, whose high order bits are the same in both operands. They assume some processors have a zero-check for operands before they are loaded from memory. This may not be true in general and it may be required to detect narrow-width operands explicitly. Their implementation detects whether a result computed by an integer function unit can be classified as a narrow-width operand. Although the area and delay overhead is reduced, if the detection is carried out in the above manner, they lose out on exploiting narrow operands that are loaded from the instruction and data cache.

The implementation by Choi et al. exploits those narrow-width operands, whose high order bits are the same [7]. The detection of narrow operands is performed explicitly by a detection logic before operands are passed on as inputs to the integer function unit. Although this ensures exploitation of all narrow operand sets, there is a significant delay and area overhead due to the detection logic.

Byte encoding applies the concept of narrow-width operands at a finer granularity with regard to the operand value. A byte wise partitioning, although chosen arbitrarily, performs well and significant power savings are reported in terms of reduction in switching activity in the ALU. Frequent-valued operands are detected by a detection logic that encodes bytes of an operand when they are loaded from the instruction cache and the data cache.

3.2 Avoiding computation

Bypassing computation is achieved efficiently by memoization. If an operand set is found to be present in the execution cache, the inputs to the function unit remain unchanged and thus switching activity through the function unit is considerably reduced. The result, which is also stored along with the operand set in the execution cache, is forwarded as the output of the function unit. The area overhead is due to the storage of the operand set and the result in the execution cache. A shortcoming of the scheme as implemented by Azam et al. is that an execution cache is required for each function unit [2]. This is an expensive overhead in terms of area and power consumed. Sharing an execution cache across multiple function units by using multiple ports would be a possible solution to this. But this would cause a severe slowdown. Adding an extra stage to the processor pipeline is also an overhead, as the overall latency increases.

In the narrow-width operand exploitation technique, the partition point for a function unit is selected based on the number of operand sets that can be exploited and the size of the function that will be active during a narrow-width operation so as to achieve significant power savings [3]. Although, this is a reasonable way to determine the partition point, it does not ensure maximum power savings. By selectively latching only the high order bytes of narrow operands, switching activity in the function unit that operates over the high order bytes of the operands is reduced. They, however, do not discuss any integer function unit in detail with regard to its structure (for example, CLA has a tree structure, ripple carry adder has a linear structure, array multiplier has an array structure, etc.). The result for the high order bits is provided through a multiplexor for frequent operands.

The partition point as determined by Choi et al. aims to maximize power savings from estimations of power savings [7]. However, they assume that power dissipation scales linearly with the size of an integer function unit. This may not always be true for different structures of the same function unit. The output for the high order bits is provided by sign extending the low order part of the result computed for the narrow operands. They use a sign extension logic to sign extend the narrow result. The high order bits of the result are provided through a multiplexor and the selection is based on a select signal produced by the detection logic. This adds to the delay and area overhead.

The byte-serial implementation as described by Canal et al. enables independent operation of an integer function unit over individual bytes of operands [8]. Thus the byte encoding technique exploits more cases of frequent valued operands as compared to the narrow-width exploitation technique. But, several dependencies between operand bytes have to be checked for to determine whether computation over the operand byte can actually be avoided. The result for an operand byte is provided by either one of the operand bytes or multiplexing hard wired values. The dependencies between operand bytes of the operand set are resolved using simple logic that varies for different function units. This also contributes to the area

and delay overheads in addition to that incurred by latching of operand bytes and multiplexing of output bits.

3.3 Estimation of power savings

The energy savings reported for the *memoization* technique are based on the assumption that the 2n-bit comparator used to detect frequent-valued operand sets consume 75% of the energy of the n-bit carry-save adders in the multiplier and the energy required to drive the word and bit lines of the 2n-bit tagged cache is approximately the same as that dissipated in an n-bit adder of the multiplier. This may report reasonable power savings at a high level of abstraction, but it does not provide realistic power savings.

The narrow-width operand exploitation technique as implemented by Brooks et al. assumes that the power dissipation of function units scales linearly at the bit level. This may not necessarily be true as function units are implemented in various structures and thus the power savings reported by them, though reasonably accurate at an architectural level, may not be very realistic.

Choi et al. implemented their scheme on the array multiplier and report power savings at the VLSI level. However, the operands for which the array multiplier was tested by Choi et al. were application-specific [7]. Such operands may not arise in general purpose processors. Also, they do not implement their technique on various integer function units for comparison.

Canal et al. estimate power savings based on activity reduction, which is based on the extension bits [8]. This does not accurately capture true power savings. They also do not explain how such a design could be extended to multiplication and division, which are more complicated as compared to addition, comparison, and bit-wise logical operations. In the parallel compressed pipeline [8], they do not explain how dependencies over different bytes of the ALU are resolved in the same cycle of operation. The detection of dependencies would require more logic, which would be different for different functions as the dependencies would vary, and thus might not be shared across different integer function units.

4 Simulations

In this section, we first describe the simulator used to obtain operand traces and the benchmarks used in our simulations. Next, we describe the simulations that were performed at the VLSI level to obtain our results.

4.1 Simulation for operand analysis

We use a version of SimpleScalar's sim-outorder to collect operands [4]. SimpleScalar provides a simulation environment for out-of-order processors with speculative execution. We collected operands for 10 million committed instructions for 5 integer (INT) benchmarks from the SPEC CPU2000 suite after skipping through a warm-up window of 500 million instructions for most benchmarks [15]. We collect 32-bit operands just before they are issued to various integer function units. The configuration of the simulator we used is given in Table 1.

Benchmarks	gzip, bzip2, gcc, mcf, gap
Compiler	gcc 2.7.2.3 for the PISA version of SimpleScalar
Inputs	Reference inputs for each benchmark
Instruction Count	10 million after the warmup window for each benchmark
Processor core	RUU size 16 instructions, LSQ size 8, Fetch queue size 4 inst/cycle, Fetch width 4 inst/cycle, Decode width 4 inst/cycle, Issue width 4 inst/cycle, Functional units 4 integer ALUs, 1 integer multiply/divide unit
Branch prediction	2048-entry bimodal predictor
L1 instruction cache	512K, 32-byte blocks, direct-mapped
L1 data cache	128K, 32-byte blocks, 4-way associative
L2 cache	Unified, 1024K, 64-byte blocks, 4-way associative

Table 1: Simulation setup to collect operands.

4.2 Simulation of integer function units

We designed and simulated the integer function units using Cadence simulation tools and Spectre simulator. We used 0.18 micron technology to design the circuits and designed them as static CMOS circuits. This type of circuit level simulation had not been done previously to determine power savings in integer function units for most of the schemes. The integer function units we designed are carry look-ahead tree adder, array-multiplier, comparator and bit-wise logical function units (AND, OR, XOR, NOR). This choice of function units covers various functions implemented in various structures. We chose the carry look-ahead tree adder and comparator to represent the tree structure, the array multiplier to represent an array structure, and the bit-wise logical function units that are implemented in a linear fashion. Our aim was to see the effectiveness of different techniques when applied to different functions implemented in different structures.

We simulated each circuit for a conventional design of the functional unit for a set of 500 inputs per bench-

mark to obtain the energy consumption in each function unit for comparison. We then simulated function units with same set of operands after modifying the design according to each scheme. Ideally, a choice for the size of the set of inputs would be as large as possible, but this would lead to excessively long simulation times.

For the memoization technique, we determined the set of operands that would be bypassed and the set of operands that would not be bypassed for a direct-mapped cache. We simulated the integer function units for those operand sets that would not be bypassed owing to their availability in the execution cache. The execution cache that we used had 64 entries and was a direct-mapped cache. We also performed simulations for the comparisons made between operand sets to determine whether an operand set was present in the execution cache. We assumed that the energy required to drive the word and bit lines of the 2n-bit tagged cache is the same as the average energy dissipated in an n-bit adder of the multiplier based on the assumption made by Azam et al [2]. We do not simulate the bit-wise logical function units as the energy consumed by the overheads would be greater than the savings that would be made if the computation were to be bypassed. This is due to the simplicity of the structure of bit-wise logical function units.

For the narrow-width operand exploitation technique, we determined the partition point in two ways. A partition point is determined so as to exploit the maximum number of operands for all the integer function units. We also determined the partition point by estimating the maximum power savings that are possible for all the integer function units [7]. We obtained partition points using both techniques and partitioned operands so that the lower part of the operand was 16 bits and 14 bits wide, respectively. In order to determine these partition points we assumed that power scales linearly at the bit-level for all the function units. We perform operand detection before the operands are issued to a function unit. The basic building blocks for the integer function units such as the full adder, carry save adder, and basic logic gates were designed as static CMOS circuits [11].

In order to implement the byte encoding technique, we simulate the byte serial implementation as described by Canal et al. for the adder, comparator, and the bit-wise logical operations [8]. Dependencies between bytes of the operands are resolved using simple logic, which varies for different operations. We extend the *byte encoding* technique to exploit operands in the array multiplier. We divide the array multiplier into four parts, so that a byte in one of the operands (A) is multiplied by all the bytes in the other operand (B). The presence of insignificant bytes in operand B is also exploited. If a byte in operand B consists of all zeros, a logical zero value is hardwired to the output of that block. We only exploit those bytes whose bits are all zero. The encoding of bytes for these operands is also modified. Each byte whose constituent bits are all zero is encoded as 1, and 0, otherwise.

5 Observations and analysis of simulation results

From our simulations at the VLSI level, we observe that the estimated power savings are highly optimistic. Actual power savings are smaller and vary for the various techniques. One of the reasons for this could be that while estimating power savings, leakage power is not taken into consideration. Another reason could be that the estimations are based on the switching activity that could potentially be reduced. This may not accurately indicate realistic power savings. Also, the assumption that power scales linearly at the bit-level may not always hold true for different function units.

Our simulations show that memoization is not very effective when applied to different integer functions. It is effective only for long latency and complicated functions. Although the presence of operand sets that are reused by the same function units is abundant (Fig. 4), exploiting these operand sets for simple functions such as addition, comparison, and bit-wise logical operations does not prove to be beneficial. We only present plots for addition, comparison, and multiplication as the overheads would be too large to exploit bit-wise operations by this technique. Memoization provides power savings only for multiplication (Fig. 7). The area overhead is huge, about 65%. The delay overhead is due to the additional stage in the processor pipeline.

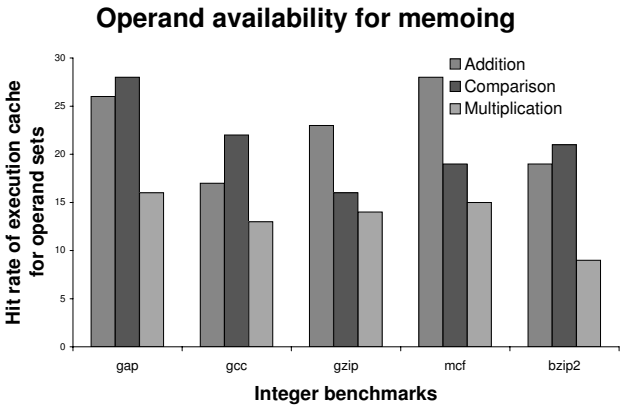


Figure 4: Availability of operands that can be exploited by memoization.

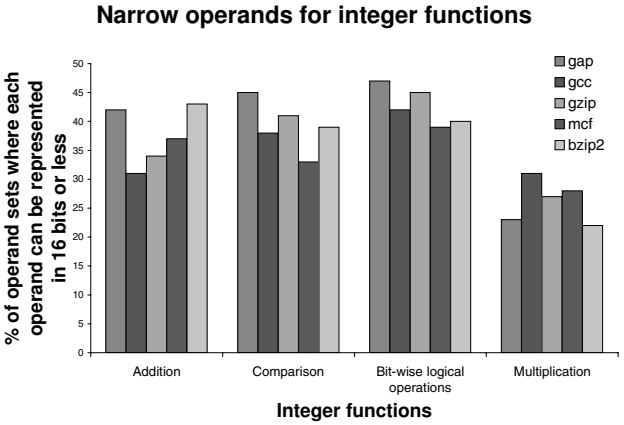


Figure 5: Availability of narrow-width operands.

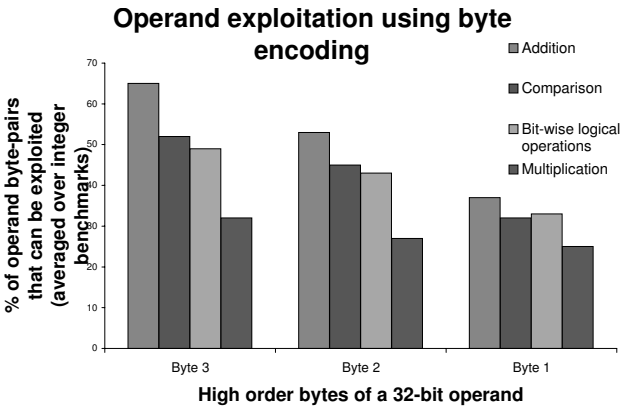


Figure 6: Availability of operands that can be exploited using byte-encoding.

The abundance of narrow-width operand sets for addition, comparison, and bit-wise logical operations is evident from Fig. 5. We also observe that narrow operand sets for multiplication are not as many. A possible explanation for this is that results from multiplication are reused as operands for multiplication frequently. In contrast, our simulations for actual power savings for the narrow-width operand exploitation technique as illustrated in Fig. 8 show that the array multiplier gives the greatest amount of power savings. This can be attributed to the fact that the array multiplier has the largest and most complicated structure as compared to the other function units under consideration. Power savings for the adder and the comparator are similar for this technique. The power savings for the bit-wise logical operations are not very significant, and this is probably due to the simplicity of the structure of these function units. We also observe that the power savings obtained by partitioning along the partition point obtained so as to maximize power savings proves to be more effective as compared to that obtained to exploit a large number of operands. The delay overhead is similar for all the operations except for multiplication. The same holds true for the area overhead.

We analyzed operand availability for exploitation by byte encoding at a byte level, since our implementation is of a byte-serial nature. Since it is possible to operate over bytes of an operand independently, we study the percentage of operands available for exploitation in individual bytes of the operand (Fig. 6). We observe that the opportunity to exploit operands is greatest in the most significant byte and decreases as we move toward the least significant byte. The power savings using byte encoding is again greatest for multiplication(Fig. 9). The area overhead is much greater compared to the narrow-width operand exploitation technique, as more logic is required to detect dependencies between operands. Also, additional logic is required to produce the output for frequent-valued operand bytes. The delay overhead is greater due to the excessive logic used.

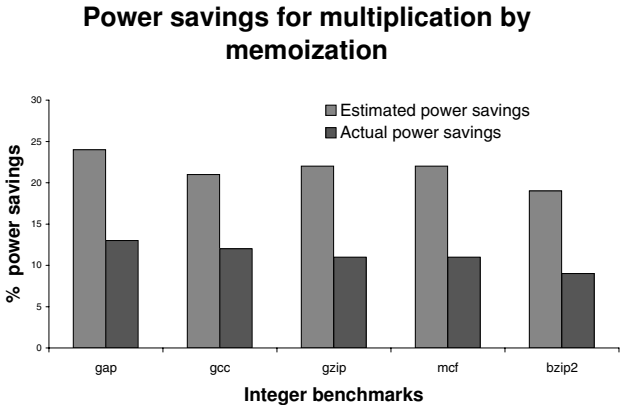


Figure 7: Power savings using memoization.

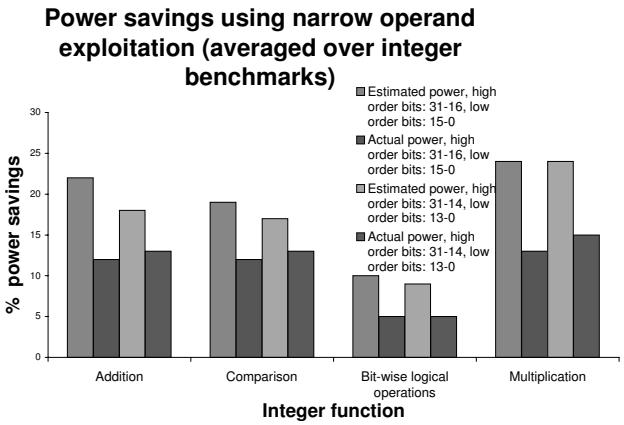


Figure 8: Power savings using narrow-width operand exploitation.

Our results show that the byte encoding technique is the most efficient technique to reduce power consumption. This is because byte encoding exploits the

Operand exploitation technique	Integer function unit	Area overhead	Delay overhead	Estimated power savings	Actual power savings
Memoization	CLA	-	-	-	-
	Comparator	-	-	-	-
	Bit-wise logical operations	-	-	-	-
	Array multiplier	65%	-	20.5%	11.2%
Narrow operand exploitation (16-bit)	CLA	22.5%	1.3%	22.2%	12.1%
	Comparator	22.2%	1.3%	19.4%	12.1%
	Bit-wise logical operations	21.9%	1.3%	10.3%	5.1%
	Array multiplier	27.4%	1.7%	24.1%	13.2%
Narrow operand exploitation (14-bit)	CLA	21.3%	1.3%	18.4%	13.4%
	Comparator	23.3%	1.3%	17.6%	13.3%
	Bit-wise logical operations	20.4%	1.3%	9.1%	5.3%
	Array multiplier	25.3%	1.7%	24.1%	14.9%
Byte encoding (byte-serial)	CLA	26.6%	2.2%	27.2%	14.2%
	Comparator	26.1%	2.1%	22.2%	14.2%
	Bit-wise logical operations	24.7%	2.2%	13.2%	6.1%
	Array multiplier	28.6%	2.4%	29%	16.4%

Table 2: Results averaged over integer benchmarks for different integer function units using various techniques to exploit frequent-valued operands.

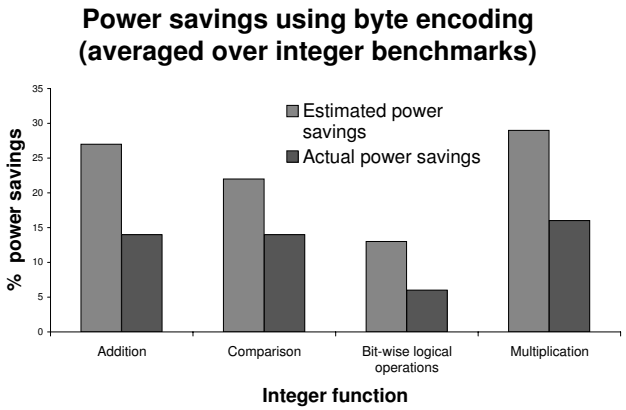


Figure 9: Power savings using byte-encoding.

most number of cases of frequent valued operands at a finer granularity. Table 2 tabulates the results. Our results show that power can be reduced by 10-15% for addition, comparison, and multiplication. The power savings for bit-wise logical operations are still lower.

6 Conclusions and future work

In this paper, we presented power savings for integer function units that were simulated at the VLSI level using various low power techniques that exploit frequent operands. We find from our results that the power estimates predicted by previous work are optimistic, whereas actual power savings are smaller. Another shortcoming to previous techniques is that they only target reduction of dynamic power dissipation. These techniques do not address the issue of leakage power, which becomes a significant part of the total power dissipated as design moves towards sub-micron technology. In order to gain considerable power savings, these techniques need to be applied more aggressively to design function units. We observe from our results that memoization, when applied to integer function units for general applications, does not perform too well. However, narrow-operand exploitation of operands and byte encoding prove to be effective. A better organized execution cache that is more efficient than the present implementation to buffer operand sets and their results could lead to greater power savings. Encoding of operand sets that are operated on repeatedly can also be considered to improve this technique. In order to exploit frequent-valued operands more effectively, a more refined partitioning of function units, based on operand statistics, that exploits more cases of frequent valued operands needs to be developed. An exhaustive enumeration of all the possible frequent valued operands that could be exploited by careful analysis of the structure of function units can also be considered for future work.

References

[1] M. Alidina, J. Monteiro, S. Devadas, A. Ghosh and M. Papaefthymiou. Precomputation-based sequential logic optimization for low power. In *IEEE Trans. Very Large Scale Integr. Syst.* 2, 4 (Dec. 1994), 426-436.

[2] M. Azam, P. Franzon and W. Liu. Low-power data processing by elimination of redundant computation. In *proceedings of the international symposium on low power electronics and design,* 1997.

[3] D. Brooks, M. Martonosi. Value-based clock gating and operation packing: dynamic strategies for improving processor power and performance. In *ACM Transactions on computer systems*, Vol. 18, 2. May, 2000.

[4] D. Burger, T. Austin. The simple scalar tool set. Tech. Rep. TR-1342. Computer Science Dept., Univ. of Wisconsin, Madison. 1997

[5] A. P. Chandrakasan, R. W. Brodersen. Minimizing Power Consumption in Digital CMOS Circuits. *Proceedings of the IEEE*, Vol. 83: 498-523, April 1995.

[6] A. P. Chandrakasan, M. Potomac, R. Mehra, J. Rabaey, R. W. Brodersen. Optimizing Power Using Transformations. In *IEEE Transactions on Computer-Aided Design*, 1995.

[7] J. Choi, J. Jeon, K. Choi. Power minimization of functional units by partially guarded computation. *ACM/IEEE International Symposium on Low Power Electronics and Design* (ISLPED), 2000.

[8] R. Canal, A. Gonzalez, J. Smith. Very low power pipelines using significance compression. *Proceedings of the 33rd Annual ACM/IEEE international symposium on Microarchitecture*, 2000.

[9] M. Pedram. Design technologies for Low Power VLSI. In *Encyclopedia of Computer Science and Technology*, Vol. 36, Marcel Dekker, Inc., 1997, pp. 73-96.

[10] M. D. Powell, S-H. Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar. Gated Vdd: A Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memories. In *ACM/IEEE International Symposium on Low Power Electronics and Design* (ISLPED), 2000.

[11] J. Rabaey. Digital integrated circuits, a design perspective. Prentice Hall, Inc. 1996.

[12] S. Rele, S. Pande, S. Onder, and R. Gupta. Optimization of Static Power Dissipation by Functional Units in Superscalar Processors. *International Conference on Compiler Construction*, LNCS 2304, Springer Verlag, pages 261-275, Grenoble, France, April 2002

[13] S. E. Richardson. Caching function results: faster arithmetic by avoiding unnecessary computation. Technical report, Sun Microsystems Laboratories, 1992.

[14] K. Roy. Leakage Power Reduction in Low-Voltage CMOS Design. In *IEEE International Conference on Circuits and Systems*, pages 167-173, 1998.

[15] K. Skadron , P. Ahuja , M. Martonosi , D. Clark. Branch Prediction, Instruction-Window Size, and Cache Size: Performance Trade-Offs and Simulation Techniques. In *IEEE Transactions on Computers*, v.48 n.11, p.1260-1281, November 1999

[16] V. Tiwari, S. Malik, P. Ashar. Guarded Evaluation: Pushing Power Management to Logic Synthesis/Design. *IEEE Trans. on CAD*, vol. 17, no. 10, Oct. 1998, pp. 1051-60.

[17] V. Tiwari , D. Singh , S. Rajgopal , G. Mehta , R. Patel , F. Baez. Reducing power in high-performance microprocessors. *Proceedings of the 35th annual conference on Design automation conference* p.732-737, June 15-19, 1998.

[18] Q. Wang and S. Vrudhula. Static Power Optimization of Deep Submicron CMOS Circuits for Dual V_T Technology. In *International Conference on Computer-Aided Design* (ICCAD), pages 490-496, 1998.