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.

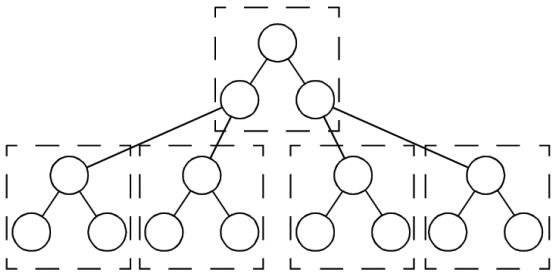

Figure 1. How nodes can be cache-consciously allocated in blocks to improve locality, (e.g. the next list node in a linked list or the children of a node in a tree)

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);
  }
}
```

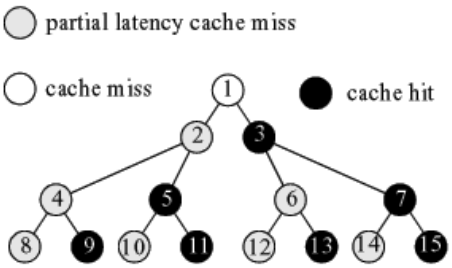

Figure 3. An example of how prefetch affects cache misses with the greedy algorithm, picture by [15]. The first node always gives a miss, if latency is two cycles. By prefetching both children the penalty will decrease for the following references.

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

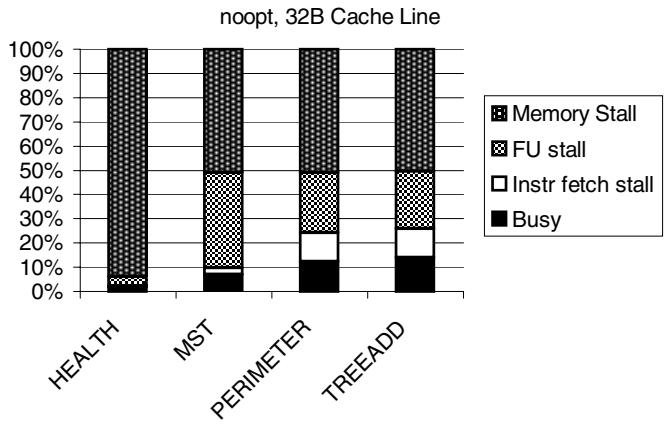

Figure 4. Stall times without cc-allocation or prefetching for the applications, on a 32 byte cache line

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

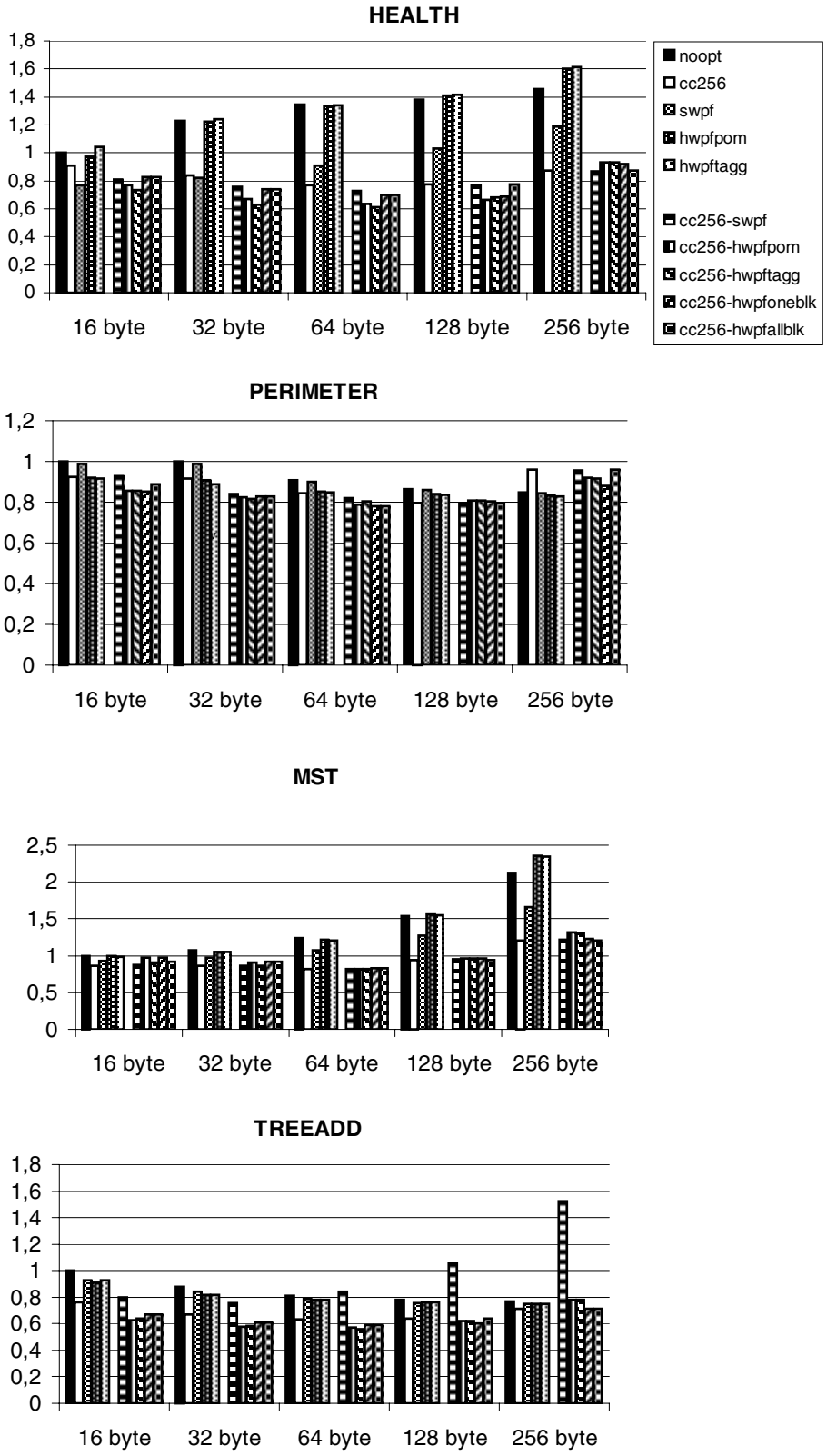

Figure 5. Normalized execution times for the applications, and all the various allocation and prefetch strategies for different cache block sizes.

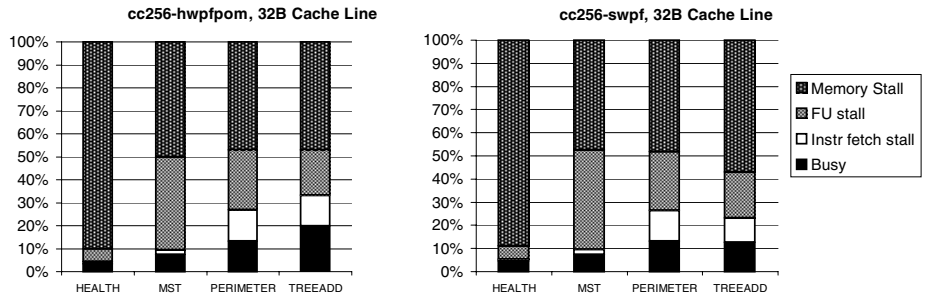

Figure 6. Stall times for cc-allocation combined with software prefetch (a) and hardware prefetch-on-miss (b)

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.

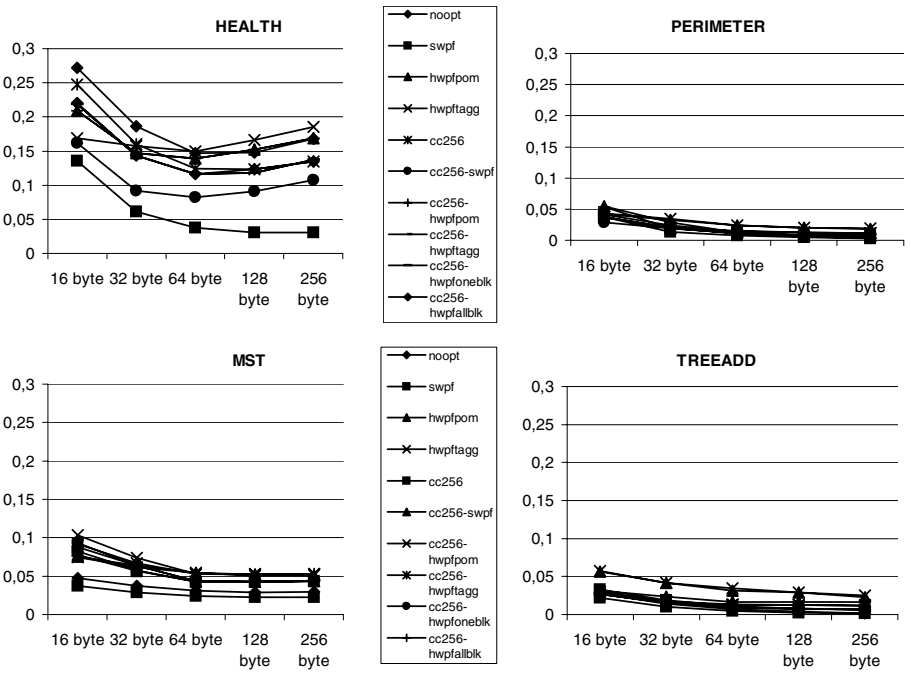

Figure 7. Level-1 data cache miss-rates for the applications and the various allocation and prefetch strategies

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].

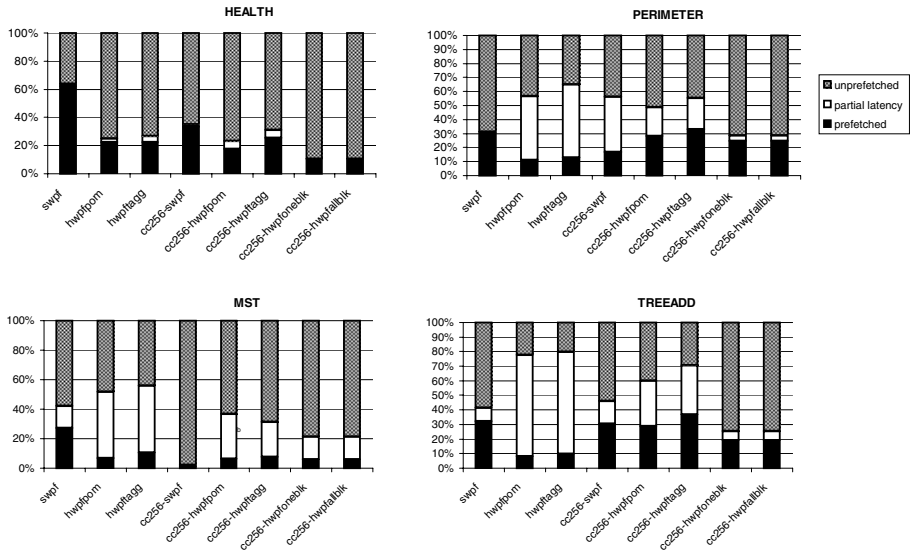

Figure 8. Prefetch Efficiency. The graphs show the fraction of loads that could be successfully prefetched, and the partially hiding, and the latency and unprefetched loads.

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.