CACHE-CONSCIOUS ALLOCATION OF POINTER-BASED DATA STRUCTURES

Angad Kataria, Simran Khurana
Student,Department Of Information Technology
Dronacharya College Of Engineering,Gurgaon

***Abstract*- Hardware trends have produced an increasing disparity between processor speeds and memory access times 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.**

I. INTRODUCTION

As processor speeds are increasing and programs are becoming more memory intensive, memory access times are a bottleneck for performance. 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, , and cache-conscious allocation of data,. 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, to limit the cost of the extra instructions. Cache-conscious allocation with a software prefetch scheme is evaluated in . 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.

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.

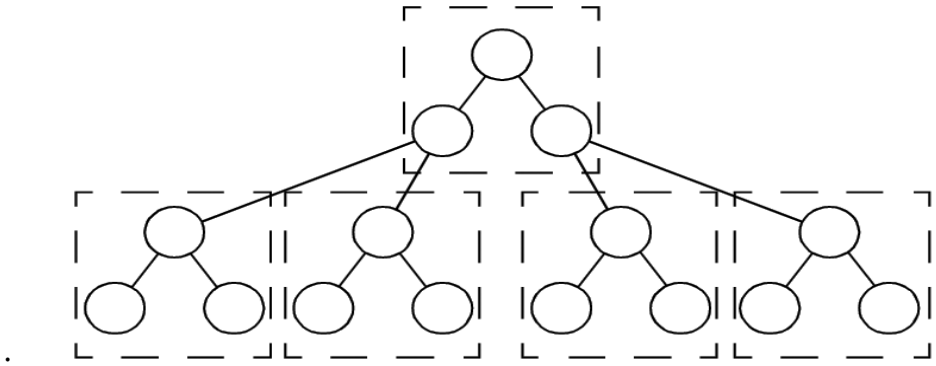

Figure 1. How nodes can be cache-consciously allocated in blocks to improve locality, (e.g. the next list node in a linked list or the children of a node in a tree)

II. CACHE-CONSCIOUS ALLOCATION

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.

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.
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.

III. PREFETCH

Prefetching structures before they are referenced will reduce the cost of a cache miss.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.
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. 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. 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.

3.2 Hardware Controlled Prefetch

There are several ways of implementing hardware prefetch support, and the algorithm choosing the lines to prefetch. Depending on the algorithm mused, 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. 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.

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

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.

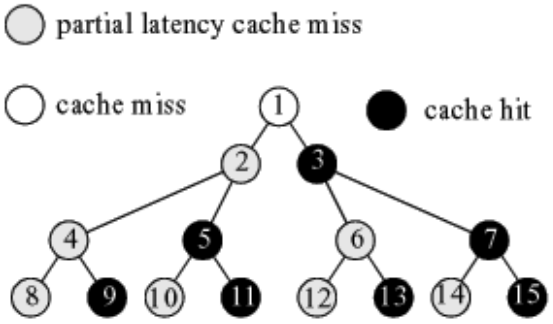

Figure 3. An example of how prefetch affects cache misses with the greedy algorithm,. The first node always gives a miss, if latency is two cycles. By prefetching both children the penalty will decrease for the following references.

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, 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 ccblocks. 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.

IV. EXPERIMENTAL FRAMEWORK

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

Hardware Architecture

The tests were conducted on an out-of-order, MIPS-like, uniprocessor simulator based on the SimpleScalar , 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.

V. CACHE PERFORMANCE

The miss-rates are improved by most optimization strategies, charts showing their improvement. 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.

5.1 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 line. 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 Hardware Prefetch combined with Cache-Conscious Allocation

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

VI. 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.

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 hotdata 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 cacheconscious data placement. In *Proceedings of the first international symposium on Memory management*, pages 37–48. ACM Press, 1998.