Compilers, Microprocessors, and Memory Systems

Preston Briggs
Keith D. Cooper

Rice University
Houston, Texas 77251-1892

1 Introduction

One key performance problem in today's microprocessor-based computers is the shifting balance between the speed of a floating-point multiply instruction and the speed of a load instruction. A decade ago, load was fast and multiply was slow. That situation has reversed over the last five years. Within a couple of years, we may see systems with processors clocked at three nanoseconds talking to fifty or seventy nanosecond memory chips. To make effective use of these fast processors, the combined hardware-software system must hide this speed mismatch. That will require changes in our compilers, our operating systems, our microprocessors, and our systems architecture.

This white paper tries to assess the state of compiler techniques for hiding memory latency. It looks briefly at trends in memory system design. Finally, it suggests a strategy for deploying our resources, in both software and hardware, to improve our ability to manage latency in real computations.

2 What can compilers do today?

Recent years have seen a large body of work developed around the problem of providing adequate performance on cache-based systems. This section tries to assess what's possible in compiler-based cache management today.

Any attempt to improve cache behavior through compile-time techniques must begin by trying to understand when cache misses occur. Several groups have looked at this problem. Porterfield *et al.* proposed a simple scheme for discovering the *overflow iteration* – the iteration of a loop where a cache miss must occur [7, 1]. Mowry *et al.* use another scheme to estimate where cache misses must occur [6]. Both techniques rely on dependence analysis to spot temporal reuse. Spatial locality is usually found by looking at loop strides. Gannon and Jalby proposed a technique based on reference windows that is an alternative to dependence-based techniques [3]. Their reference windows allow them to notice spatial locality, too.

Discovering where to improve the code is just the first step. To make the code run faster, we must transform it so that its behavior better maps onto the memory hierarchy. Work in this area falls into three major categories: blocking, copying, and prefetching.

2.1 Cache blocking

Cache blocking (also called *tiling* or *strip mining*) is a technique for reshaping the iteration space of a loop to improve its locality [9, 10, 8, 2]. Some loops overrun their reuse – they would reuse values from cache except that those values have been knocked out of the cache by intervening references. Blocking improves the performance of these loops by bringing references to a single location closer together in time. While the specific transformations involved in blocking varies from paper to paper, they include loop interchange, strip mining, loop skewing, and loop reversal.

Blocking improves performance in one specific case: when reuse exists, but the combination of cache and program behavior results in replacement of data before its reuse. Such premature replacement comes from several sources:

1. Alignment of data can cause interference (both within a single data structure and between multiple structures).

This research is supported by ARPA through ONR grant N00014-91-J-1989.

Warning: This paper contains strong opinions on hardware. We have drawn these opinions from reading and listening. We have not done the extensive simulation studies that would be required to assess completely the practicality of our ideas.

2. The number of items used between successive references can exceed the cache's effective size, ensuring replacement before reuse.

3. The replacement algorithm implemented in the cache can be a poor match to the program's actual behavior.

Blocking directly addresses the second problem. By moving successive references closer together, it increases the likelihood of successful reuse – reuse within the cache. As a side effect, it may decrease the likelihood of the other two problems, but this is an indirect effect. Blocking changes neither alignments nor replacement policies; it reshapes the iteration space.

2.2 Copying

Copying [4] (or *streaming* [5]) attempts to improve behavior by treating the cache as a fast local memory. In effect, the generated code treats the cache as a single large array. Values are explicitly copied into the array before use and modified values are copied back to main memory after their last reuse. This lets the code sidestep problems of alignment, stride, and replacement policy; the compiler tries to dictate all data movement in and out of the cache. As a further, often ignored benefit, copying can remove TLB misses from inner loops – making performance far more stable.

Copying imposes some overhead. Lam *et al.* show that the benefits of copying often exceed the overhead [4]. To minimize the overhead, the map from subscripts in the original code to subscript in the pseudo-array used to represent cache should be simple. A further problem arises if the instructions used to load and unload the cache allocate cache lines themselves. This introduces a subtle problem with self-interference – the loads required to copy the data into cache can cause replacements of data already moved into the cache.

2.3 Prefetch (and flush)

Both blocking and copying work within a given architecture. The use of an advisory prefetch instruction offers an interesting alternative, albeit one that requires carefully designed hardware support.

Conceptually, advisory prefetch is simple. We add an additional instruction to the architecture – `prefetch` ⟨*expr*⟩. It initiates a fetch of the cache line containing ⟨*expr*⟩ into cache. Barring resource constraints, this allows the compiler to overlap execution with the time required for the fetch operation – a clear win. The principal drawback of an advisory prefetch scheme is that it requires modifications to the hardware.

Researchers have shown that even simple schemes for inserting prefetch instructions can be surprisingly effective [6, 1]. The decreased time spent waiting for a fetch to complete often more than compensates for the additional instruction issue slots required. A corresponding `flush` instruction might be used to control replacement when there is a mismatch between the hardware-enforced policy and the actual reference patterns.[1]

2.4 Assessment

Each of these techniques addresses the problem of mapping program locality onto hardware cache structures. Each has strengths; each has weaknesses. Compilers should use these techniques whenever possible; the payoffs are large enough to compensate for the extra complexity in the compiler.

Nonetheless, questions remain about the effectiveness of these techniques on production codes. The examples shown in papers and talks are almost all small loop kernels. The extent to which these techniques can improve production applications remains to be seen.[2] Today, we are basically limited to improving the memory performance of simple loop nests.

[1]Remember, LRU replacement is not necessarily what we want. It is simply a compromise based on statistical properties of programs. For a specific program, other replacement policies may work better.

[2]A particular concern is that real loops reference more aggregate data items that the associativity of real caches. Loop distribution may help in such cases – when it is safe and legal.

3 Trends in hardware

Compilers must target specific architectures. A decade ago, compilers largely ignored specific details of the memory hierarchy. Today, these details are critical. Thus, we should look briefly at trends in the design and construction of memory systems. We will focus on cache issues, but register sets and TLBs are equally important.

Trends in cache design are hard to decipher. New machines have large caches (PA-RISC), small caches (ALPHA & i860), direct-mapped caches (SPARC & PA-RISC), and set-associative caches (RS/6000 & i860). They implement diverse replacement algorithms: pseudo-random (i860XP), pseudo-LRU (i486), and LRU (RS/6000). Some even provide `prefetch` instructions (ALPHA).

Several larger trends do emerge, however. Cache lines are getting longer. Primary caches, in general, become larger across different generations of the same architecture. Many systems are being designed with secondary caches (SPARC-10 & ALPHA-based systems). The one dimension that is not growing is set associativity – there is a fundamental conflict between updating the context required for LRU replacement and increasing associativity. If, in fact, a general trend exists in associative cache design, it is away from real LRU replacement.

Each of these trends is an attempt to increase the likelihood that a given value resides in cache. If small caches are good, larger caches should be better. There is substantial justification for this reasoning. Unfortunately, most of the arguments in favor of these trends ignore one fundamental fact – in many cases, the compiler can analyze reference patterns, recognize broad styles of locality, and provide the hardware with hints.

4 A modest proposal

In the semiconductor industry, it is widely accepted that the number of devices on a single integrated circuit roughly doubles every eighteen months. Naturally, this trend has provoked speculation about how to use the massive number of transistors available on a single chip by the end of the decade. Current trends would move in the direction of larger on-chip caches and integration of multiple chips on a single die.

With all this extra chip real estate available, we propose taking a different approach to designing a memory hierarchy. The trends cited in Section 3 can be viewed as "more of the same." Simply put, larger caches should produce higher hit ratios. Unfortunately, this strategy produces diminishing returns – it requires ever more cache for an ever smaller improvement in the hit ratio. The other trends – longer lines, relaxing replacement policies – simplify the construction of larger caches.

Instead of building larger caches, we propose that new architectures provide hardware support for other kinds of locality. The support for `prefetch` in the ALPHA is a step in this direction. We should take it further. Rather than providing one principal path from main memory to the register set, we should provide several paths with different properties. For example, we should consider

- hardware support for gather and scatter operations between a small local memory and main memory. The success of copying (see Section 2.2) argues for this support.
- a small cache (short lines & large associativity) to support references that exhibit temporal locality without spatial locality, like pointers. Such a cache differs from a register in that it provides address resolution. Analysis of addresses at compile-time *is* imprecise.
- a small, fully-associative cache with compiler-controlled replacement. Self-interference due to alignment is a problem with small caches; it cannot be sidestepped easily. Let the compiler worry about replacement with `flush` instructions.
- support for `prefetch` and `flush` in *each* level of the hierarchy. Such support might include a separate issue slot dedicated to `prefetch` and `flush` operations.
- non-allocating load and store instructions for two reasons: (1) to handle values that exhibit no reuse, and (2) to let programs avoid easily detectable forms of self-interference. For example, the `pfld` operations on the i860XP have proven useful to the Portland Group's compiler [5].

- programmable *fifo*'s like those proposed in Wulf's WM architecture [11]. These can be viewed as sophisticated, programmable gather-scatter operations for non-reused values; an alternative view is that they are tools to let the compiler avoid generating prolog and epilog loops
- hardware support for coherence between these structures, where possible.

Of course, none of these make sense without a smart compiler. The classic primary cache cannot simply disappear, unless we mandate improvements in basic compiler technology. That seems an unlikely scenario.

In short, compilers can discover different styles of locality. Current cache structures support several of these well, but fail to help with others. Further, by forcing all references through a single structure, we decrease the cache's effectiveness on those references where it could help most. We should build hardware that offers different paths from main memory to registers – paths that match the different styles of locality known to a modern compiler. This lets the compiler provide the hardware with contextual information about locality – information encoded in the choice of instructions.

References

[1] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Systems*, April 1991.

[2] Steve Carr. *Memory-Hierarchy Management*. PhD thesis, Rice University, Department of Computer Science, September 1992.

[3] Dennis Gannon, William Jalby, and Kyle Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, 5:587–616, 1988.

[4] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architecural Support for Programming Languages and Operating Systems*, April 1991.

[5] Larry Meadows, Steven Nakamoto, and Vincent Schuster. A vectorizing, software pipelining compiler for LIW and superscalar architecture. In *Proceedings of RISC '92*, San Jose, CA, February 1992.

[6] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. *SIGPLAN Notices*, 27(9):62–75, September 1992. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems.*

[7] Allan Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications.* PhD thesis, Rice University, Department of Computer Science, May 1989.

[8] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, June 1991.

[9] Michael Wolfe. Iteration space tiling for memory hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, December 1987.

[10] Michael Wolfe. More iteration space tiling. In *Proceedings of the Supercomputing '89 Conference*, 1989.

[11] William A. Wulf. The WM computer architecture. *Computer Architecture News*, 16(1):70–84, March 1988.