The Declining Effectiveness of Dynamic Caching for General-Purpose Microprocessors

Douglas C. Burger, James R. Goodman, Alain Kägi

Computer Sciences Department
University of Wisconsin-Madison
1210 West Dayton Street, Madison, Wisconsin 53706 USA
galileo@cs.wisc.edu

Abstract

The computational power of commodity general-purpose microprocessors is racing to truly amazing levels. As peak levels of performance rise, the building of memory systems that can keep pace becomes increasingly problematic. We claim that in addition to the latency associated with waiting for operands, the bandwidth of the memory system, especially that across the chip boundary, will become a progressively greater limit to high performance. After describing the current state of microsolutions aimed at alleviating the memory bottleneck, this paper postulates that dynamic caches themselves use memory inefficiently and will impede attempts to solve the memory problem. We present an analysis of several important algorithms, which shows that increasing levels of integration will not result in computational requirements outstripping off-chip bandwidth needs, thereby preserving the memory bottleneck. We then present results from two sets of simulations, which measured both the efficiency with which current caching techniques use memory (generally less than 20%), and how well (or poorly) caches reduce traffic to main memory (cache sizes up to 2000 times worse than optimal). We then discuss how two classes of techniques, (i) decoupling memory operations from computation, and (ii) explicit compiler management of the memory hierarchy, provide better long-term solutions to lowering a program's memory latencies and bandwidth requirements. Finally, we describe Galileo, a new project that will attempt to provide a long-term solution to the pernicious memory bottleneck.

1 Introduction

The growing inability of current memory systems to keep up with processor requests has significant ramifications for the design of microprocessors in the next decade. Baskett recently estimated [5] that the annual rate of performance increase for single-chip microprocessors is 80% (roughly 5% per *month*). Extrapolating this rate ten years into the future suggests single-processor performance of roughly 30,000 Spec-marks by the year 2005.

While it is not unthinkable that such processing power might be achieved in this time frame, the authors believe that other limitations on performance are likely to emerge that render the achievement of such raw processing power irrelevant. The design of a memory system capable of satisfying such voracious processors will become one of the key technical challenges facing system designers. We postulate that processors will eventually be designed to be just fast enough so that they can consume operands as fast as the memory system can produce them. Currently, the opposite is true: the memory system is designed to keep up with the processor. Performance of the system will then be largely determined by (1) how effectively the on-chip memory is able to maintain operands, minimizing the frequency of off-chip access, and (2) the rate at which the external memory system can supply operands.

Presently, the most serious concern with memory supplied from off-chip is latency: the increasing clock rate of emerging processors and the ability to issue multiple instructions per cycle means that even the fastest off-chip access is growing to scores or even hundreds of instruction issue times. Much research in computer architecture today is focused on minimizing or tolerating this latency. We anticipate that some of the techniques being researched will be successful in eliminating at least a large fraction of the latency. Such a success will then expose a deeper and more fundamental limit: the raw bandwidth achievable from outside the chip (or module).

It is our contention that the current memory system model (dynamic caches sitting in between the processor and large, off-chip main memories) will be fundamentally incapable of supporting long-term solutions to either the latency or the bandwidth problems. In this paper, we survey current techniques being explored to tolerate latency through prescient requests for needed data. In the context

This work is supported in part by NSF Grant No. CCR-9207971 and an unrestricted grant from Apple Computer Inc.

of these techniques, we make arguments as to why the bandwidth bottleneck will eventually arise, and perform an analysis of several algorithms that supports our argument. We then present experimental evidence that bolsters our hypothesis, measuring both the effectiveness with which caches use their memory, and the gap between the bandwidth requirements of a cache and a near-optimal memory of the same size.

The effective use of local (on-chip) memory will be essential for alleviating the memory bottleneck. We develop the notion of *memory efficiency* [34, 46], to measure how successfully current microprocessors make use of their local memories (which are primarily cache memory). We will use this measure to demonstrate that current microprocessors use their on-chip memory very poorly, suggesting that alternative ways of managing the on-chip memory may be far more effective than cache memory.

Our presentation then turns to a discussion of several techniques that we believe have much more potential for providing extremely fast future processors with a sufficient stream of operands. Finally, we introduce *Galileo*, a project dedicated to finding long term solutions to this problem, and propose an advanced system that is the first step in our search for a solution to the memory bottleneck.

The reminder of our paper is organized as follows: Section 2 describes the memory bottleneck in great detail and presents arguments supporting the formation of a long-term bandwidth bottleneck. Section 3 discusses current caches in this context, describing why they have been so successful and why we believe they will ultimately be limited. Section 4 presents a couple of alternatives to the current memory system organization, and introduces the architecture that we hypothesize may eliminate memory system limitations for ultra-fast processors. Section 5 draws final conclusions from this work and describes the direction in which our work is headed.

2 The memory bottleneck

As new fabrication technologies mature, feature sizes decrease, reducing capacitance and permitting faster switching of gates. As the feature size λ drops, the surface density of transistors increases as $O(\lambda^{-2})$. These trends have produced CPUs with ten million transistors and DRAMs with 64 MB per chip. To effectively utilize these processors, designers have been moving more and more high-speed memory onto the processor chip, memory that can effectively use the tremendous intra-chip bandwidths available.

Unfortunately, not all physical trends follow these curves. Off-chip bandwidth grows slowly, as packaging technology matures. The cost per pin is dropping far more slowly than the cost per transistor. The rate at which data can be moved across these pins is also increasing more slowly, as are the bandwidths of inter-chip connections. Access times for DRAMs are diminishing more slowly than are cycle times for CPUs, at the annual rate of 5-10% [1]. Jointly, these factors produce the frequently-expounded-upon *memory bottleneck*, in which accesses to main memory, in terms of processor cycles, become progressively more expensive as the “5% months” pass.

2.1 Characterizing memory delays

The execution of a task can be roughly decomposed into periods according to which component of the system limits the speed of execution. We divide the task’s execution into being *processor-bound* and *memory-bound*. A task is processor-bound when the pipeline is not stalled waiting for an operand from memory, and memory-bound when it is.[1] A memory-bound phase of execution can be subdivided into periods of being *latency-bound* or *bandwidth-bound*. Latency-bound describes a task that is stalled because a memory operation was not issued sufficiently ahead of the operand’s first use. A task is bandwidth-bound when it contains sufficient parallelism to prevent stalling for a contention-free memory access, but contention for memory increases the access latency enough to cause a processor stall.[2]

2.2 Current solutions

Current methods for reducing stall time associated with main memory accesses fall into two broad categories: (i) *reducing* latency, and (ii) *tolerating* latency.

Techniques for reducing the mean access latency for memory operands can be subdivided into two classes: reducing the *number* of access penalties, and reducing the *length* of the access penalty. Memory access penalties can be avoided by keeping live values (that will be referenced more than once) physically close to the processor. Caches are the ubiquitous technique for doing so. By automatically moving missing references into a local (i.e., on-chip) memory, a cache exploits temporal locality when another reference to that word is issued later. By using cache lines that are larger than a word, a cache implicitly performs transparent, speculative prefetching of the words around the missed address, attempting to exploit spatial locality. By increasing the size of an on- or off-chip cache, more live values can be simultaneously held near the processor, reducing the number of requests issued to main memory. Techniques that improve a cache’s hit ratio (e.g., higher

1. Another definition for processor bound: if doubling the clock cycle time doubles the execution time of a phase of computation, it is processor-bound.
2. Alternatively, a phase is bandwidth-bound if providing more memory bandwidth would eliminate processor stalling.

associativity and better replacement policies) tend to reduce the number of main memory requests. While caches are commonly thought of as latency-reducing units, local memories in general, and caches in particular, have also been identified [19] as a way of reducing the bandwidth requirements. Traffic to the next lower level of the memory hierarchy is reduced by the cache servicing some fraction of the requests at the higher level.

Techniques for reducing the actual main memory latency include faster interconnects, more levels of off-chip caches, and reduced access times for DRAMs [26, 49, 37]. Bus clock speeds and data widths are increasing. Page-mode DRAMs reduce the access latency to multiple adjacent memory references [33]. Decreasing feature sizes are driving down DRAM cycle times, by 5-10% [1] per year.

Techniques for tolerating latency have been widely studied, and there is still considerable ongoing research in this area. We partition these techniques into two classes: *scheduling* and *prefetching*. Scheduling attempts to overlap a memory access with other useful instructions, exploiting available instruction-level parallelism. Prefetching techniques, conversely, provide non-binding hints to the memory system that attempt to bring imminently referenced data closer to the processor.

In order for memory accesses to be scheduled so that useful work may be performed in parallel, memory operations must be non-blocking. Lockup-free caches [29] allow multiple memory requests to be outstanding, and compilers attempt to place as many instructions as needed between a non-blocking load and the first use of its result. The issuing non-blocking loads can be thought of as a first step toward decoupling memory accesses from computation. Gharachorloo, Gupta and Hennessy [18] have explored the use of dynamically scheduled processors to hide memory latencies. While their exploration was performed in the context of shared-memory multiprocessors, some of their conclusions are more broadly applicable.

Prefetching techniques consist of both hardware and software prefetches. Hardware prefetching typically uses dynamic stride detection to perform run-time calculation of prefetch addresses to be issued [3, 15, 16]. The overheads of hardware prefetching are the cost for the additional hardware, and the limited ability of the dynamic units to perform any prefetching other than through arrays with linear strides. A different form of hardware prefetching consists of stream buffers [27, 36]. Chen and Baer [8] evaluated the effectiveness of lockup-free caches and hardware prefetching, and proposed a hybrid scheme based on a combination of these approaches.

Software prefetching is much more flexible than hardware prefetching, having the advantage of compile-time knowledge, but pays the price of software overhead, both in instructions issued and code size [7, 28, 35]. Chen et al. [10] examined the trade-offs between prefetching data directly into the cache and prefetching into a prefetch buffer. The lack of run-time knowledge can also be an impediment to purely software techniques. Some promising approaches use hybrid hardware/software techniques, issuing limited instructions that provide hints to the prefetch hardware [11]. Chen and Baer [9] studied software and hardware prefetching schemes in the context of a multiprocessor and proposed a hybrid approach combining software and hardware schemes.

2.3 The bandwidth bottleneck

We believe that these substantial efforts to reduce latencies of contention-free memory references will ultimately be successful, even for less regular non-scientific codes. The relative success of this effort will result in the exposure of a more fundamental limit to performance, the memory bandwidth bottleneck. Two factors will drive this exposure. Increasingly faster program execution, with progressively fewer stalls waiting for operands, will eventually reach the point where requests to memory are issued faster than they can be handled. Also, techniques used to reduce memory latencies, such as speculative loads and prefetches, will trade increased bandwidth consumption for lower latency, hastening the point at which congestion in the memory system becomes severe. Providing sufficient off-chip and memory bandwidth to supply a high-speed future processor with a poorly functioning on-chip memory will not be a cost-effective solution. We now show that neither the latency reduction nor the latency tolerance techniques already discussed will prevent programs from becoming bandwidth-bound.

2.3.1 Reducing memory access latency

Reducing the physical time needed for a main memory access can increase the bandwidth to main memory. However, DRAM access times are dropping much more slowly than those of CPU clocks—on the order of 5-10% per year [39, 14, 1]. The rate of increase of processor pins is much slower than that of transistor density. Although there are significant breakthroughs in packaging technology on the horizon, the issues of reliability, power, and cost will prevent pins from sustaining growth commensurate with the rate of transistor increase. The speed of light is also a fundamental limit to the amount by which off-chip access latencies can be reduced. The union of these trends and limits indicates that physical increases in off-chip bandwidth are unlikely to match increases in processing capability, particularly for when cost is accounted.

2.3.2 Reducing frequency of memory accesses

Increasing levels of integration will permit substantial

Algorithm	Memory	Computation	Memory traffic	Processing/ traffic ratio
TMM	$O(N^2)$	$O(N^3)$	$O(N^2/\sqrt{S})$	$\sqrt{f}$
Stencil	$O(N^2)$	$O(N^2)$	$O(N^2/\sqrt{S})$	$\sqrt{f}$
FFT	$O(N)$	$O(N\log_2 N)$	$O(N\log_2 N/\log_2 S)$	$\log_2 f$
Sort	$O(N)$	$O(N\log_2 N)$	$O(N\log_2 N/\log_2 S)$	$\log_2 f$

Table 1: Application growth rates [30]

increases in on-chip memory. It is possible that such increases will reduce memory traffic requirements enough so that many algorithms that were formerly memory-bound become processor-bound. More local memory will enable greater reuse of operands, which will reduce the traffic to main memory required by the program. This will raise the ratio of computation to memory traffic.

Over time (measured in months or years), technological advances will permit faster processing and a higher bandwidth across the chip boundary. The ratio of the two quantities is the processing capability per unit of off-chip traffic due to technological advances. Assume that some computation is bandwidth-bound, and that the change in the latter ratio as time progresses is less than the change in the ratio of computation to memory traffic (due to more on-chip memory). The increased amount of computation made possible by the increase of on-chip memory will then outstrip the rate at which the processor is getting faster (per unit of memory traffic), and the bandwidth-bound computation becomes processor-bound. Conversely, if the processor speed increases at a sufficient rate to ensure that it can always handle the increased availability of on-chip operands, then the computation will remain bandwidth-bound.

Naively, one would expect the processing requirements to eventually overwhelm the bandwidth limitations, since for many algorithms the computation grows faster than do the memory requirements. For example, the conventional algorithm of matrix multiply (multiplying two $N \times N$ matrices) has total memory requirements that grow as $O(N^2)$, while computation grows as $O(N^3)$.

This simplistic argument is misleading. Consider the conventional matrix multiplication, using a tiled algorithm where tiles are of size $T \ll N$. It is easily shown [22, 32] that the traffic between the on- and off-chip memory is proportional to $2N^3/T + N^2$. Assume that the processor is sufficiently fast for the algorithm implemented to take full advantage of the on-chip memory. Holding N constant keeps the amount of computation constant. If the on-chip memory is increased, less memory traffic is required, allowing the program to complete in less time. An increase in the on-chip memory (which is of size S) by a factor of f would increase T by $\sqrt{f}$, which would reduce the off-chip traffic by $O(\sqrt{f})$. Therefore, the execution will still be bandwidth-bound if the processor speed is also increased by a factor of $\sqrt{f}$. The rate of increase in processor speed need be proportional only to the square root of the rate of increase of local memory size for the program to remain bandwidth-bound.[1] Table 1 shows such derivations for the following algorithms: TMM (tiled matrix multiply), Stencil (an algorithm operating on a $N \times N$ matrix, which repeatedly updates each element with a weighted sum of neighboring elements), FFT (an N-point fast Fourier transform), and Sort (merge sort). The right-most column depicts the change in the ratio of computation to required memory traffic for each application, as S is increased by a factor of f. If this quantity grows faster than the processing speed as S increases, a computation will eventually become processor-bound. However, the minimum rate of processing speed increase necessary to maintain the bandwidth bottleneck of a computation is certainly less than existing rates of processor improvement. Increasing the local memory will therefore *not* solve the bandwidth problem.

2.3.3 Tolerating latency

Lockup-free, or non-blocking, caches are an important device for tolerating memory access latencies. While non-blocking caches make feasible the elimination of one source of memory latency (stall on every miss), they may create another source of latency: contention in the memory

1. Whether external caches count as local memory depends on whether the metric of interest is traffic across the processor pins or traffic seen at main memory.

system. The current solutions being proposed to tolerate memory latencies, described in Section 2.2, allow memory operations to be overlapped with other instructions, including other memory references. This increases the bandwidth load on the memory system by trying to move the same number of requests across the chip boundary in a shorter time.

Optimizations that better tolerate memory latencies, in addition to increasing the density of memory requests, frequently *increase* the total bandwidth requirement of a program, trading greater bandwidth needs for reduced latencies. Prefetches can increase bandwidth requirements, particularly when they are issued too early, by evicting needed data in the cache that would have been used before the prefetched data was needed. Prefetched data may also be evicted from the cache by another reference before it is actually used, forcing it to be re-fetched. A more severe bandwidth increase occurs with speculative prefetches and speculative loads. When the compiler has an insufficient number of instructions within a basic block to hide the memory latency, the prefetch or load may be *lifted* [39] across a branch, becoming speculative. Speculative memory operations that turn out to be unnecessary will thereby increase the total bandwidth requirement of a program.

Latency tolerating techniques will generally only increase the bandwidth load on the memory system, not reduce it. We have shown that for at least four algorithms, increasing the on-chip memory to reduce the average memory operation latency will *not* eliminate a bandwidth bottleneck over time. We have also described qualitatively why decreasing the total memory bandwidth requirements, by reducing the number of off-chip accesses, will not outstrip the increased bandwidth requirements due to faster processors. This leads us to conclude that memory bandwidth will emerge as a major new limitation to system performance as the fraction of uncongested miss latencies that stall the processor diminishes. Current techniques will not alleviate, but will more likely exacerbate, the pending situation.

3 Evaluating the efficacy of caches

Caches are becoming almost ubiquitous in the microprocessor industry. Every major general-purpose, commodity processor recently announced has an on-chip cache. The community of scientific machine designers has traditionally been reluctant to use caches, and many of the canonical scientific supercomputers used no caches at all [40]. Said designers are increasingly being forced to design machines with caches, however, as cost/performance curves increasingly mandate the use of general-purpose processors.

3.1 Why caches?

Caches have become so prevalent primarily because they are transparent; they generally offer visible, and sometimes substantial, performance improvement while requiring no support from the compiler, programmer, or architecture.[1] Caches take advantage of both temporal and spatial locality. They can satisfy multiple references to the same word, and perform implicit prefetching by implementing block sizes that are larger than one word. They can thereby eliminate most references to main memory. They have also been extensively studied, and are consequently very well understood. Voluminous papers examining block sizes, associativities, replacement policies, virtual versus physical addressing, and unification versus separation of instruction and data caches have appeared in the literature [41, 42].

3.2 Evaluating local memory efficiency

Caching as a technique has also been so successful because it is effective at reducing latency penalties and memory system bandwidth. Given the rate at which memory penalties are increasing, however, it is becoming progressively more important to use the available amount of local memory as well as possible. In this section we characterize how well a given local memory (cache) is used.

We define the *efficiency* of a given memory to be the average fraction of the memory that holds “live” data [34, 46] at any point in the execution of the program. A frame in the cache is defined to be *live* if its contents will be read again before they are written; i.e, a block’s lifetime extends from its first write until the last time it is read.

We have modified DineroIII [21] and used Shade [12], a tracing tool from Sun, to produce measurements of a cache’s efficiency. Live time is considered to be the time between a store and a read hit, or two consecutive read hits. We measured “liveness” on the granularity both of individual words and of cache blocks, which were assumed to be 16 bytes (with 4-byte words) for the purposes of this study. We varied cache sizes, simulating cache sizes of all powers of two between 4KB and 2MB, inclusively. We simulated caches with set associativities of 1, 2, and 4. The experiments were run on the following benchmarks: buk, compress, eqntott, g++, su2cor, and swm256. Buk is a NAS [4] kernel that implements bucket sort. G++ is release 2.6.0 of the Gnu C++ compiler. It generated the assembly code of the preprocessed CPU module of a multiprocessor simulator, and was run with full optimization enabled. Compress, eqntott, swm256, and su2cor are all from the Spec92 [50] suite. Compress and eqntott

1. This is not to say that such support has never been provided; it is simply not required for correctness.

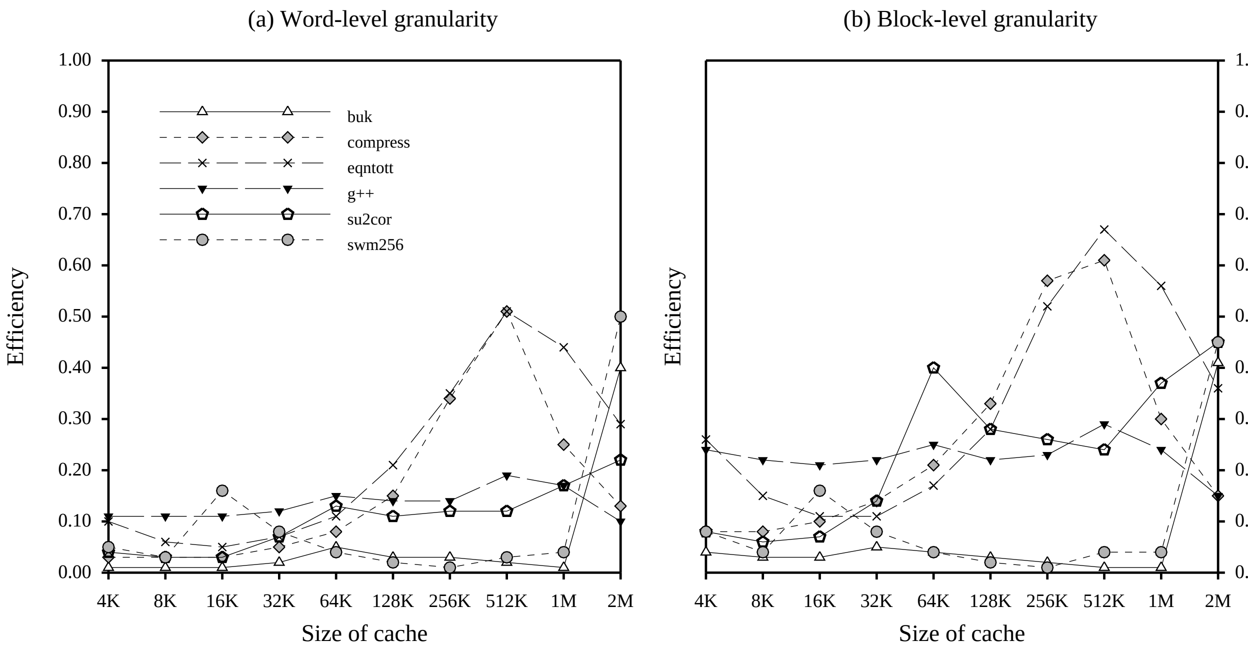

Figure 1. Efficiency measurements

were run with the default inputs. Su2cor was run with a short input, and swm256 was run with the default input for 20 iterations. All benchmarks were traced on Sun Sparcstation 10 workstations, and compiled with **`-O3 -mflat`** using GCC version 2.6.0. The “mflat” option compiles code without using the SPARC register windows. Using register windows would have hidden a fraction of the addresses produced by the benchmark code from our trace, as instructions from traps on window overflows and underflows are not output by Shade. The libraries we used were unavoidably compiled with register windows, and therefore generated some addresses that were not included in our trace.

Figure 1 plots the measured efficiency of 4-way set-associative caches for sizes from 4KB to 2MB. Figure 1(a) depicts efficiencies calculated by differentiating between live and dead words within blocks, and Figure 1(b) shows cache efficiencies examining blocks rather than individual words (e.g., the period in between two references to different words in the same line would be considered live). For the most part, our simulations follow a predictable trend. Caches substantially smaller than the data set size (and/or the working set size) of the traced application show poor efficiency, as loaded lines are evicted after few uses and the cache thrashes. Efficiency improves with increasing cache size, peaking at the point when the working set, and occasionally the entire data set, fits in the cache.

The numbers obtained by varying set associativity did not deviate substantially from the results already presented. Higher set associativities tended to produce slightly better efficiencies, although we observed a few slight exceptions. We present the efficiency numbers for 4-way associative caches only, to present the most optimistic results. In order to establish that the low efficiencies were due to poor use of the cache, and not cold-start or dead-data (at program termination) effects, we measured the dead time before a frame’s first and after a frame’s last reference. Those quantities were appropriately negligible, indicating that the programs were sufficiently long-running to prevent endpoint effects from biasing our results.

What is surprising about these numbers is not the shapes of these curves but the scales; the efficiencies tend to remain under 20% for cache sizes that are much smaller than the data set size of the applications. The word-level efficiency numbers are even worse; they tend to be lower than 10% for caches smaller than the applications’ data sets. The ratio between the block- and word-level efficiencies gives a rough idea of what percentage of the words in the block are actually used. Codes that access data structures linearly, and with a stride of one, will tend to produce similar efficiencies for the two experiments; this is clearly demonstrated by the numbers for swm256, the one array-dominated code depicted. In general, the word-level efficiencies should always be less than block-level efficiencies, since if any word in a block is live, block-level runs count the whole block as live. The one data point where this relation does not hold is swm256 with a 2MB cache. The block-level efficiency is lower than the word-level efficiency here because of the way liveness is calculated for blocks; a store marks everything in the block as dead. In this particular case, blocks that contained multiple live words were declared dead in the block-level calculation, enough that the block-level efficiency was driven under the word-level efficiency.

Although the efficiencies for the larger caches tend to be high compared to those for the smaller caches, these are uninteresting data points because of the close correspondence between the larger cache sizes and the applications' data sets. Our benchmarks were the limiting factor at this point, and were we to perform runs on much larger applications, the efficiencies for 256KB-1MB caches would be as low as the small cache efficiencies displayed in Figure 1. The two benchmarks that support this claim are swm256 and buk, both of which have larger data sets than the other benchmarks. Swm256 has a data set of about 4M, and buk's is roughly 6MB. Both programs display efficiencies of under 5% for a *one megabyte* cache. As with the other benchmarks, efficiency rises precipitously when the cache is sufficiently large to hold the working set, which for these two benchmarks is about 2MB. Buk and swm256 efficiencies for a 4MB cache decline from those of a 2MB cache (the 4MB results were not graphed). Even when the cache size is closest to the working-set size, the highest word-level efficiencies were just above 50%, which is a poor best-case utilization.

3.3 Reducing memory bandwidth

The existence of such low efficiencies for interesting workloads demonstrates that the potential exists for much better use of local memory than a cache. Given our hypothesis that memory bandwidth will become a critical limitation, we ran a series of experiments designed to measure the gap between the memory traffic generated by a cache and that generated by an unreasonably well-managed local memory.

The latter memory unit, to which we will refer as **n-opt**, is simulated as a fully-associative (level-one) cache that has a block size of one word. The replacement policy uses **MIN** [6], in which dead cache blocks are always victimized. If every block in the cache is live, the cache block that will be referenced farthest in the future is chosen as a victim. This policy is called **n-opt** (near-optimal) because **MIN** is not optimal; in some cases it is preferable to evict a clean block that will be referenced sooner, rather than a dirty block that will be referenced later [23]. Since **MIN** assumes a perfect oracle, however, and is unlikely to be realized in the near future, **n-opt** is a sufficient bound for our study.

To obtain these results, we used Shade [12] to generate an address trace in an appropriate format. The traces were then fed to Cheetah [51], with which we computed miss ratios for both the **n-opt** model and a normal (L1) cache, using a least-recently-used replacement policy with set associativities of 1 and 4 (**lru-1** and **lru-4**). The lru block sizes were 16 bytes. We chose a smaller line size so as not to bias our results in favor of **n-opt**. Based on the number of cache misses, we calculated the total amount of data moved in between the cache and main memory. We were forced to ignore write-back traffic, since our memory traffic numbers were calculated from the miss rates of the caches.

Figure 2 graphs the total memory traffic (program I/O) generated for the different cache models. The traffic, in megabytes, is presented as a function of cache size for compress, g++, eqntott, and swm256. The results for buk are similar to those for eqntott, as they both perform heavy sorting, and are not presented here. As with the efficiency experiments shown in Figure 1, we consider only the cache sizes that are substantially smaller than the data set sizes, roughly 128KB and smaller, to be interesting, but we present the larger numbers for completeness.

For compress, **n-opt** reduces the bandwidth generated over that of both **lru-1** and **lru-4** by roughly a factor of 5. **N-opt** does a much better job of keeping the heavily accessed parts of the 64KB hash table resident in the local memory. The bandwidth for g++ is similarly reduced by a factor between 2 and 3. G++ is characterized by a large, chaotically accessed data set, portions of which display moderate temporal locality. Hence there is no sharp drop-off in bandwidth requirements for any sizes or models, just a gradual lessening of memory traffic as more and more of the data set fits in the local memory. Eqntott and swm256 demonstrate a similar pattern: small local loops or working sets cause thrashing for **lru-1** and **lru-4**, but not for **n-opt**. Once the local memory is sufficiently large, the lack of large-scale temporal locality in these codes (some in eqntott and virtually none in swm256) causes **n-opt** and **lru** to perform similarly.

Small **n-opt** memories reduce the memory traffic to surprisingly low levels, particularly when compared against the size of an **lru-4** cache necessary to produce an equivalent amount of traffic. Example comparisons are shown by the dotted lines in Figure 2. The quotients of the **lru-4** size divided by the **n-opt** size for the examples shown are approximately 2000, 60, 250, and 130, for (a), (b), (c), and (d), respectively. These examples corroborate the poor efficiency shown in Section 3.2.

The relative flatness of the **n-opt** curves shows that for memory that is managed very well, surprisingly small amounts will greatly reduce the memory traffic. Although **n-opt** represents an unrealizable point, the memory size differentials needed to generate the same amount of traffic are so large that less aggressive schemes should provide substantially better-used memory.

3.4 Why not data caches?

The results in Section 3.2 demonstrate that data caches use memory inefficiently (lowering their effective size)

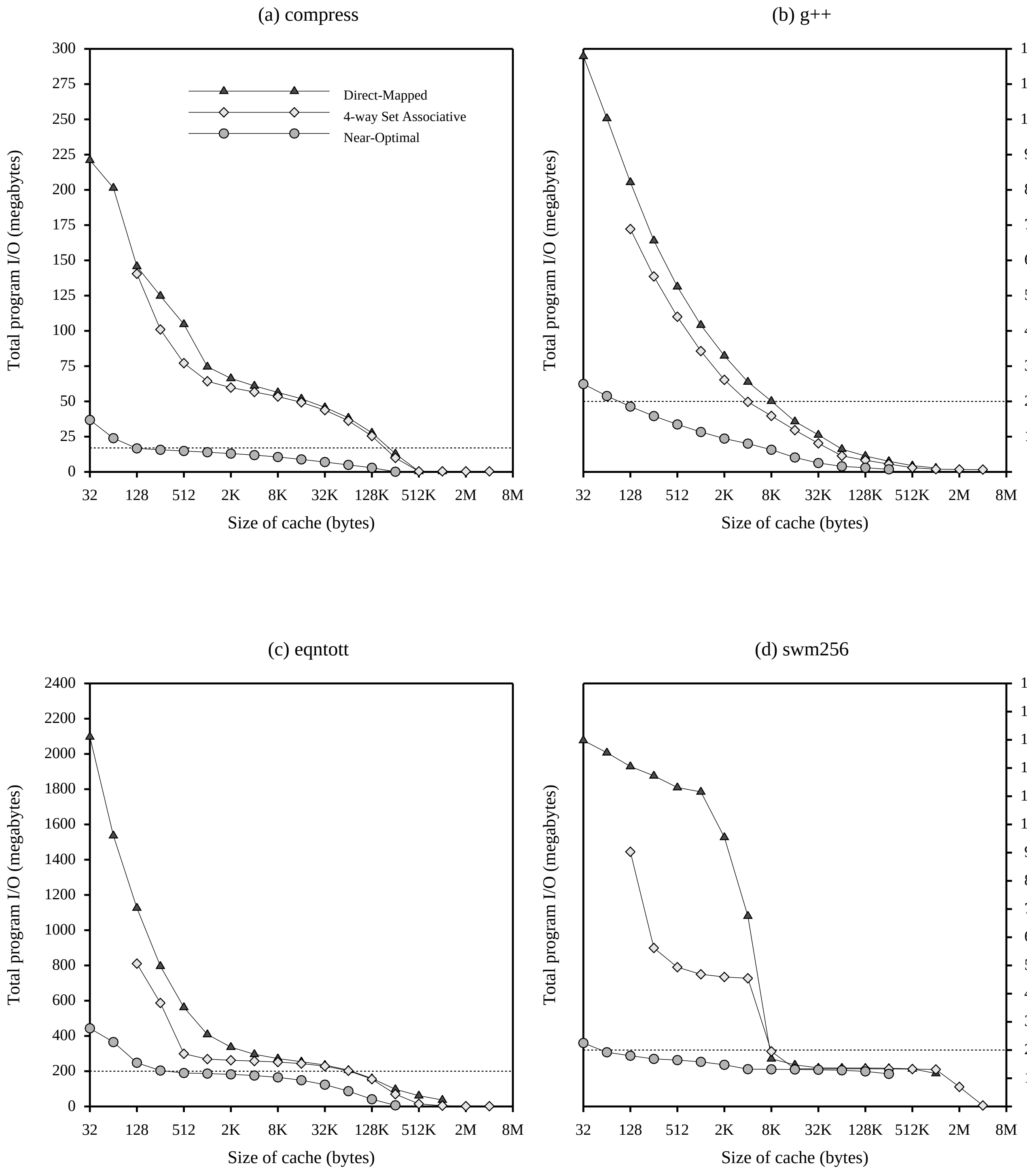

Figure 2. Memory bandwidth differentials

when a program does not fit entirely in the cache. Section 3.3 showed that near-optimally managed local memory generates the same amount of memory traffic as caches that are from one to three orders of magnitude larger (even without prefetching, etc.). In a bandwidth-bound system, where the effectiveness of local memory must be maximized in order to minimize memory traffic, data caches perform extremely poorly.

It is precisely the generality of current caches that will limit their effectiveness for future microprocessors. Their transparency comes at a cost; preventing the system from treating them as a resource to be carefully managed. Both programmer knowledge and compile-time information are unavailable to caches, which are currently incapable of using such information anyway. Cache organizations are fixed, limited to one replacement policy, which can severely degrade performance in cases where the policy is ill-suited for the reference stream. They have limited associativity, generating conflicts when substantial numbers of dead frames exist in the cache. The block size is fixed,[1] preventing requests for larger blocks (incurring extra misses) when there is more spatial locality, and smaller blocks (consuming unnecessary bandwidth) when there is less or none. The tag storage is certainly not negligible (and is not considered in our studies presented above). The fixed binding of addresses to data in the cache prevents any sort of partitioning scheme to be performed on the cache.

Although the aforementioned problems are limiting, performance using the current caching model will quickly deteriorate once the memory access penalty grows severe enough to merit aggressive usage of the techniques described in Section 2.2. Caches are ill-suited toward supporting latency-tolerating techniques such as prefetches and speculative loading. Multiple outstanding requests can frequently return out of order. Caches have no concept of ordering, and may allow returning requests to evict one another. Some efforts to perform heavy prefetching into a cache have met with success [35], but these efforts relied on extremely regular codes with detailed and specific analyses that would be difficult to generalize. These regular codes would be quite easy to manage in software without a cache, given the appropriate hardware support.

When massive prefetching is performed, the current and future working sets may conflict. Proposals in the literature have suggested providing separate units, one to contain the current working set and the other to fill simultaneously with prefetched data [11, 14]. Physically partitioning the memory optimizes for a particular workload size, and will generally be inferior to a logical, programmable partitioning of a single unit. As more and more parallelism is exploited by future systems, careful scheduling will become even more of a necessity than it is today, and the nondeterminacy of today's caches will become prohibitively expensive.

Finally, the *allocation* of frames on a cache miss may drastically reduce cache efficiency when heavy prefetching is performed. Empty, allocated memory may be a major contributor to increased bandwidth load and poor utilization of memory, particularly in a bandwidth-bound system. Allowing slots to be allocated when data *returns* may serve as an important optimization, but this is far beyond the capabilities of current caches, and introduces difficult new problems.

In terms of software support needed (none) and hardware complexity, current caches are quite inexpensive. The performance gap between caches and more complex, expensive memory structures, however, is growing along with the memory access penalty. As this division grows, the cost of more complicated memory structures will become increasingly justifiable.

4 Long-term solutions

This work is part of project that we call *Galileo*. The goal of our project is to identify long-term, technology-driven impediments to performance, and propose solutions. In this vein, we are attempting to determine what techniques hold the best promise for providing a long-term solution to the memory bottleneck. In this section, we discuss two techniques that we feel hold promise, and then propose an architectural philosophy, based on these two techniques, that will be our project's first point of evaluation.

4.1 Decoupling

Although explicitly decoupled architectures [43, 44, 20, 47] have not achieved success in the mainstream commercial market, the philosophy of decoupling memory operations from computation is becoming increasingly visible in modern commercial processors, such as the PowerPC [48] and the MIPS TFP [24].

Decoupled architectures have several fundamental advantages. They implicitly provide aggressive latency tolerance, and permit much higher performance of the execute unit, through the use of queues and renaming. The access unit has the potential for optimizing memory accesses; this will be further discussed in Section 4.3. These architectures can be viewed as a generalization of vector processing; they lend themselves well to such computations. We believe that they hold the potential for similarly high scalar code performance.

One of the well-known penalties to which decoupled

1. Subblocking is an attempt to deal with this problem, but is only a partial solution.

architectures are susceptible are loss-of-decoupling events [43] (LODs). These events are due to data and/or control dependencies that would also cause traditional processors to incur large penalties, particularly when the memory access latency is very large. They also provide an easy point for optimization: memory latencies are seen *only* at LODs. Decoupled architectures, however, make no attempt to conserve the bandwidth to the memory system; we address this issue also in Section 4.3.

4.2 Explicit memory management

The evidence presented in this paper suggests that caches make very inefficient use of a given amount of memory. We claim that: (i) memory bandwidth will eventually become a critically limiting factor, (ii) caches do very poorly in minimizing traffic loads on the memory system, and (iii) caches will cause performance to suffer greatly in the presence of memory latency reduction techniques. In this section we explore some alternatives to simple caches for the upper levels of the memory hierarchy.

Our view is that compilers, over time, will gradually be given greater responsibilities for managing the upper levels of the memory hierarchy. State-of-the-art compilers perform sophisticated program analyses, which is usually thrown away once the code is generated, making it unavailable to the run-time system. In many cases, the hardware is forced to re-create these analyses for both correctness and better performance [45]. Substantial work has been done that tries to enable the compiler to circumvent the cache, performing analyses to work against the hardware [1, 53, 52].

Several levels of compiler control are possible for the local memory hierarchy. The compiler could issue hints (flavors) along with the memory operations, specifying whether to move data into a specific level of the cache or bypass the cache entirely (for data that will only be read or written once in a long while). These techniques would improve the efficiency of the cache, by preventing the replacement of other data that might still be live with data that will soon be dead [1].

By mapping different portions of the address space into different cache-like memory units, the compiler could place classes of data into units whose hardware policies were most suitable for the access pattern of a particular class. Different hardware units could have different capacities, access times, associativities, replacement policies, or even block sizes. Some of these units could reside closer to the datapath than the level one cache [2].

A longer-term option is to provide local memory units in addition to the caches; units that are completely managed by the compiler. Such units could reduce both latency (better hit rates) and memory traffic considerably. Data would be moved in and out of these units by the compiler, according to their access patterns. Having software control of the replacement algorithm, with the added benefit of compile-time information, could prove extremely effective. Much more precise scheduling would be possible, as the access time for data residing in such a unit would be deterministic.[1] Conflict misses would not occur, a shorter cycle time would be possible due to the absence of hit/miss detection logic, and the effective memory size would be greater due to the absence of tag storage.

Caches, as previously discussed, do not lend themselves well toward latency tolerance optimizations. Prefetched data (a future working set) can collide with the present working set in a cache, exacerbating the latency problem rather than solving it. A strong argument supporting explicitly-managed units is the fact that the compiler could logically partition the memory so as to prevent conflicts between current and future working sets of data.

Software-managed local memory does have several drawbacks, however. The most severe shortcoming is the possibility of multiple copies of the same datum simultaneously residing in different locations. It is unlikely that this disambiguation problem will ever be solved, unless programming models evolve to support a solution. One direction that holds promise for increasing the classes of variables that can be managed explicitly is run-time disambiguation [13, 17, 25]. Other disadvantages of explicit memory management include tremendous overheads for task switching, and increased instruction counts (both code size and dynamic path length).

4.3 A long-term proposal

Increasing levels of integration and improvements in packaging technology will eventually yield chips and/or multi-chip modules that have hundreds of millions of gates. System designers will have the option of adding as much processing capability as the memory system can support; that is, any additional processing power would go unutilized. The rate at which the memory system can supply operands, in turn, will be determined by off-chip traffic.

The processor appears to be infinitely fast if adding more processing capability has no effect on program execution time. The point at which this situation will be effectively reached is fast approaching. We argue that systems designed under these constraints will always be bandwidth-bound, as designers will add more processing capability if memory bandwidth is going unused.

Off-chip accesses in these systems will be so expensive that physically separating the processor and memory by a

1. A better term for such a memory might be "deterministic-access memory".

chip boundary will be counterproductive. Rather, all system memory should be coupled on-chip with a processor. We expect that feature sizes will be so small that a large portion of on-chip real estate will be dedicated to dense memory. These levels of integration will allow all system memory to be moved on-chip, changing the view of the system from processor-centric to memory-centric.

If more memory is required for a system, then more of these homogeneous processor/memory modules will be added. This will allow the system to leverage off decades of parallel processor research. Each module will be responsible for processing what is in its own local memory; processing on off-chip memory will be invoked remotely rather than having the data brought locally and processed. We note that many in the research community would describe this as coarse-grain dataflow.

The processing units on this module will be composed of a decoupled architecture. The access unit, in addition to being responsible for gathering operands to feed to the execute unit, will also manage a wide, rich memory hierarchy that will allow it tremendous flexibility in scheduling memory operations. This will permit it to optimize both on-chip memory scheduling and remote accesses. When making remote operations, the access units of this system would therefore greatly resemble hardware protocol processors from some current multiprocessor projects (Tempest and FLASH) [38, 31], although they would be much more integrated, and would function at a considerably lower level.

We also expect that the programming model and compiler technology will evolve to support a “data-pushing” model, rather than a request/response model (which may remain the worst-case default). Rather than sending requests for remote operands, an access unit would be responsible for *sending* data from its local memory that was needed by other access units. It would also be responsible for receiving and organizing data it needed from remote access units.

This model blurs the distinction between uniprocessors and multiprocessors, and between distinct processor chips and memory chips. While radical and not yet well-defined, this system would minimize the off-chip traffic, the speed of which is limited by the speed of light. The bandwidth bottleneck will disappear, lost in the tremendously wide paths of on-chip memory. Latency will be as overlapped as the programs permit, completely decoupled, with little uncertainty in the high-speed memory hierarchy. Computation would then be truly limited only by the pace of technological advance.

5 Conclusions

Today’s technological trends point to a widening gap between the rate at which a processing unit can consume operands and the rate at which the memory system can supply them. Present designs are addressing this trend by introducing one or two levels of on-chip cache. While this on-chip memory effectively reduces memory access latency, the delay incurred when it is necessary to go off-chip is high. As a consequence, processors extrapolated from current designs will be more and more frequently stalled waiting for operands.

We have argued that this problem can be alleviated by making effective use of the on-chip memory and by maximizing the effectiveness of off-chip bandwidth. We explored the concept of *memory efficiency*, to evaluate the effectiveness of memory in reducing needed bandwidth. We then showed through simulation that cache memories do not use local memory efficiently.

Most processor stalls occur today because the off-chip requests experience latency. A wide variety of latency reduction and latency tolerance techniques are being investigated, and some show great promise of reducing or eliminating stalls from failure to request operands sufficiently far in advance.

Many algorithms have greater computational requirements than memory requirements. The naive implication of this fact is that computational requirements will outstrip the bandwidth requirements given a sufficiently large on-chip memory. We demonstrated that this assumption is false by analyzing the off-chip memory accesses of blocking algorithms, demonstrating that as the cache grows larger, allowing an increase in the blocking factor, the computation ability grows at a faster rate than the increase in the block size. Thus it is reasonable to conclude that future processors will become increasingly memory-bound even if the raw processing speed increases only linearly with cache memory size.

We have proposed two techniques to address the limitations envisioned by the present trends: decoupled access/execution and explicit memory management. Decoupling memory accesses from computations minimizes the frequency of stalls due to memory latency, and provides an efficient way for overlapping memory operations. Decoupling, or other techniques for prefetching off-chip operands, will effectively solve the off-chip latency problem, enabling computational speed improvements until the more fundamental limit of off-chip bandwidth is reached.

Explicit management of the on-chip memory permits more efficient use of memory. Eliminating the cache for most memory accesses permits the compiler to control the use of memory rather than trying to anticipate what the cache might do. Prefetching, in particular, can be performed much more effectively if the compiler can fully control which operands will be on-chip and which will be off. In addition, the predictability resulting from explicit fetching, even in the presence of some variability in off-chip access time, makes scheduling of on-chip operations far more effective.

Successors to current architectures must be limited in their innovation due to compatibility expectations and other forms of market inertia. While this is undoubtedly the correct approach for development of processors that depend on commercial success for their exploration, it is important to keep in mind the longer-term effect of current trends. This work is the initial result of a new project, Galileo, which is focused on the long-term implications of changing technology and how current trends will affect architecture. The results here are only examples, but demonstrate how focusing on long-term limits may lead us in new directions, or at least help in evaluating current proposals in terms of their long-term potential.

Acknowledgments

We would like to thank Babak Falsafi, Stefanos Kaxiras, Subbarao Palacharla, Jim Smith, and T. N. Vijaykumar for their helpful discussions. In addition, we thank Babak and Steve Reinhardt for their CPU module code that we used as an input to the g++ benchmark.

References

[1] Santosh G. Abraham, Rabin A. Sugumar, B. R. Rau, and Rajiv Gupta. Predictability of Load/Store Instruction Latencies. In *Proceedings of the 26th International Symposium on Microarchitecture*, pages 139–152, December 1993.

[2] Todd M. Austin, T. N. Vijaykumar, and Gurindar S. Sohi. Knapsack: A Zero-Cycle Memory Hierarchy Component. Technical Report 1189, Computer Sciences Department, University of Wisconsin-Madison, November 1993.

[3] Jean-Loup Baer and Tien-Fu Chen. An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty. In *Proceedings of Supercomputing '91*, pages 176–186, November 1991.

[4] David Bailey, John Barton, Thomas Lasinski, and Horst Simon. The NAS Parallel Benchmarks. Technical Report RNR-91-002 Revision 2, NASA Ames Research Center, August 1991.

[5] Forest Baskett. Keynote address. *International Symposium on Shared Memory Multiprocessing*, April 1991.

[6] L. A. Belady. A Study of Replacement Algorithms for a Virtual-Storage Computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[7] David Callahan, Ken Kennedy, and Allan Porterfield. Software Prefetching. In *Proceedings of the Fourth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, April 1991.

[8] Tien-Fu Chen and Jean-Loup Baer. Reducing Memory Latency via Non-blocking and Prefetching Caches. In *Proceedings of the Fifth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 51–61, October 1992.

[9] Tien-Fu Chen and Jean-Loup Baer. A Performance Study of Software and Hardware Data Prefetching Schemes. In *Proceedings of the 21th Annual International Symposium on Computer Architecture*, pages 223–232, April 1994.

[10] William Y. Chen, Scott A. Mahlke, Pohua P. Chang, and Wem mei W. Hwu. Data Access Microarchitectures for Superscalar Processors with Compiler-Assisted Data Prefetching. In *Proceedings of the 24th International Symposium on Microarchitecture*, pages 69–73, November 1991.

[11] Tzi-cker Chiueh. Sunder: A Programmable Hardware Prefetch Architecture for Numerical Loops. In *Proceedings of Supercomputing '94*, pages 488–497, November 1994.

[12] Bob Cmelik and David Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurements and Modeling of Computer Systems*, pages 128–137, May 1994.

[13] Peter Dahl and Matthew O'Keefe. Reducing Memory Traffic with CRegs. In *Proceedings of the 27th International Symposium on Microarchitecture*, November 1994.

[14] Stefanos Damianakis, Kai Li, and Anne Rogers. An Analysis of a Combined Hardware-Software Mechanism for Speculative Loads. Technical Report TR-455-94, Princeton University, April 1994.

[15] John W. C. Fu and Janak H. Patel. Data Prefetching in Multiprocessor Vector Cache Memories. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 54–63, May 1991.

[16] John W. C. Fu, Janak H. Patel, and Bob L. Janssens. Stride Directed Prefetching in Scalar Processor. In *Proceedings of the 25th International Symposium on Microarchitecture*, pages 102–110, December 1992.

[17] David M. Gallagher, William Y. Chen, Scott A. Mahlke, John C. Gyllenhaal, and Wen mei W. Hwu. Dynamic Memory Disambiguation Using the Memory Conflict Buffer. In *Proceedings of the 6th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 183–193, October 1994.

[18] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Hiding Memory Latency using Dynamic Scheduling in Shared-Memory Multiprocessors. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 22–33, May 1992.

[19] James R. Goodman. Using Cache Memory To Reduce Processor-Memory Traffic. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 124–131, June 1983.

[20] James R. Goodman, Jian-tu Hsieh, Koujuch Liou, Andrew R. Pleszkun, P. B. Schechter, and Honesty C. Young. PIPE: A VLSI Decoupled Architecture. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 20–27, June 1985.

[21] Mark D. Hill, James R. Larus, Alvin R. Lebeck, Madhusudhan Talluri, and David A. Wood. Wisconsin Architectural Research Tool Set. *Computer Architecture News*, 21(4):8–10, August 1993.

[22] Jia-Wei Hong and H. T. Kung. I/O Complexity: the Red-Blue Pebble Game. In *Proceedings of the 13th Symposium on Theory of Computing*, pages 326–333, May 1981.

[23] L. P. Horwitz, R. M. Karp, R. E. Miller, and A. Winograd. Index Register Allocation. *Journal of the ACM*, 13(1):43–61, January 1966.

[24] Peter Yan-Tek Hsu. Designing the TFP Microprocessor. *IEEE Micro*, 14(2):23–33, April 1994.

[25] Andrew S. Huang, Gert Slavenburg, and John Paul Shen. Speculative Disambiguation: A Compilation Technique for Dynamic Memory Disambiguation. In *Proceedings of the 21th Annual International Symposium on Computer Architecture*, pages 200–210, April 1994.

[26] Fred Jones. A New Era of Fast Dynamic RAMs. *IEEE Spectrum*, 29(10):43–49, October 1992.

[27] Norman P. 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, May 1990.

[28] Alexander C. Klaiber and Henry M. Levy. An Architecture for Software-Controlled Data Prefetching. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 43–53, May 1991.

[29] David Kroft. Lockup-Free Instruction Fetch/Prefetch Cache Organization. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 81–87, May 1981.

[30] H. T. Kung. Memory Requirements for Balanced Computer Architectures. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 49–54, June 1986.

[31] Jeffrey Kuskin et al. The Stanford FLASH Multiprocessor. In *Proceedings of the 21th Annual International Symposium on Computer Architecture*, pages 302–313, April 1994.

[32] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Proceedings of the Fourth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, April 1991.

[33] Sally A. McKee and William A. Wulf. Access Ordering and Memory-Conscious Cache Utilization. In *Proceedings of the First International Symposium on High-Performance Computer Architecture*, January 1994.

[34] Geoffrey D. McNiven and Edward S. Davidson. Analysis for Memory Referencing Behavior For Design of Local Memories. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 56–63, May 1988.

[35] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of the Fifth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, October 1992.

[36] Subbarao Palacharla and R. E. Kessler. Evaluating Stream Buffers as a Secondary Cache Replacement. In *Proceedings of the 21th Annual International Symposium on Computer Architecture*, pages 24–33, April 1994.

[37] Rambus Inc. *Architectural Overview*, Mountain View, California, 1992.

[38] Steven K. Reinhardt, James L. Larus, and David A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21th Annual International Symposium on Computer Architecture*, pages 24–33, April 1994.

[39] Anne Rogers and Kai Li. Software Support for Speculative Loads. In *Proceedings of the Fifth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 38–50, October 1992.

[40] Richard M. Russel. The CRAY-1 Computer System. *Communications of the ACM*, 21(1):63–72, January 1978.

[41] Alan Jay Smith. Cache Memories. *Computing Surveys*, 14(3):473–530, September 1982.

[42] Alan Jay Smith. Bibliography and Readings on CPU Cache Memories and Related Topics. *Computer Architecture News*, 14(1):22–42, January 1986.

[43] James E. Smith. Decoupled Access/Execute Computer Architectures. In *Proceedings of the 9th Annual International Symposium on Computer Architecture*, pages 112–119, April 1982.

[44] James E. Smith. Decoupled Access/Execute Computer Architectures. *ACM Transactions on Computer Systems*, 2(4):289–308, November 1984.

[45] James E. Smith. Invited talk. *21th Annual International Symposium on Computer Architecture*, April 1994.

[46] James E. Smith. Private Communication. September 1994.

[47] James E. Smith et al. The ZS-1 Central Processor. In *Proceedings of the Second Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 199–204, October 1987.

[48] James E. Smith and Shlomo Weiss. PowerPC 601 and Alpha 21064: A Tale of Two RISCs. *IEEE Computer*, 27(6):46–58, June 1994.

[49] IEEE Computer Society. IEEE Standard for High-Bandwidth Memory Interface Based on SCI Signaling Technology (RamLink). Draft 1.00 IEEE P1596.4-199X, December 1993.

[50] Standard Performance Evaluation Corporation. *SPEC Newsletter*, Fairfax, Virginia, December 1991.

[51] Rabin A. Sugumar and Santosh G. Abraham. Efficient Simulation of Caches under Optimal Replacement with Applications to Miss Characterization. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurements and Modeling of Computer Systems*, pages 24–35, May 1993.

[52] Olivier Temam, Elana D. Granston, and William Jalby. To Copy or Not to Copy: A Compile-Time Technique for Assessing When Copying Should be Used to Eliminate Cache Conflicts. In *Proceedings of Supercomputing '93*, pages 410–419, November 1993.

[53] Michael E. Wolf and Monica S. Lam. A Data Locality Optimizing Algorithm. In *Proceedings of the 1991 Conference on Programming Language Design and Implementation*, pages 30–44, June 1991.