Memory Bandwidth Limitations of Future Microprocessors

Doug Burger, James R. Goodman, and Alain Kägi

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

Abstract

This paper makes the case that pin bandwidth will be a critical consideration for future microprocessors. We show that many of the techniques used to tolerate growing memory latencies do so at the expense of increased bandwidth requirements. Using a decomposition of execution time, we show that for modern processors that employ aggressive memory latency tolerance techniques, wasted cycles due to insufficient bandwidth generally exceed those due to raw memory latencies. Given the importance of maximizing memory bandwidth, we calculate effective pin bandwidth*, then estimate optimal effective pin bandwidth. We measure these quantities by determining the amount by which both caches and minimal-traffic caches filter accesses to the lower levels of the memory hierarchy. We see that there is a gap that can exceed two orders of magnitude between the total memory traffic generated by caches and the minimal-traffic caches—implying that the potential exists to increase effective pin bandwidth substantially. We decompose this traffic gap into four factors, and show they contribute quite differently to traffic reduction for different benchmarks. We conclude that, in the short term, pin bandwidth limitations will make more complex on-chip caches cost-effective. For example, flexible caches may allow individual applications to choose from a range of caching policies. In the long term, we predict that off-chip accesses will be so expensive that all system memory will reside on one or more processor chips.*

1 Introduction

The growing inability of memory systems to keep up with processor requests has significant ramifications for the design of microprocessors in the next decade. Technological trends have produced a large and growing gap between CPU speeds and DRAM speeds. The number of instructions that the processor can issue during an access to main memory is already large. Extrapolating current trends suggests that soon a processor may be able to issue hundreds or even thousands of instructions while it fetches a single datum into on-chip memory.

Much research has focused on reducing or tolerating these large memory access latencies. Researchers have proposed many techniques for reducing the frequency and impact of cache misses. These include lockup-free caches [28, 40], cache-conscious load scheduling [1], hardware and software prefetching [6, 7, 13, 14, 26, 32], stream buffers [24, 33], speculative loads and execution [11, 35], and multithreading [30, 38].

This work is supported in part by NSF Grant CCR-9207971, an unrestricted grant from the Intel Research Council, an unrestricted grant from the Apple Computer Advanced Technology Group, and equipment donations from Sun Microsystems.

It is our hypothesis that the increasing use and success of latency-tolerance techniques will expose memory bandwidth, not raw access latencies, as a more fundamental impediment to higher performance. Increased latency due to bandwidth constraints will emerge for four reasons:

1. Continuing progress in processor design will increase the issue rate of instructions. These advances include both architectural innovation (wider issue, speculative execution, etc.) and circuit advances (faster, denser logic).
2. To the extent that latency-tolerance techniques are successful, they will speed up the retirement rate of instructions, thus requiring more memory operands per unit of time.
3. Many of the latency-tolerance techniques increase the absolute amount of memory traffic by fetching more data than are needed. They also create contention in the memory system.
4. Packaging and testing costs, along with power and cooling considerations, will increasingly affect costs—resulting in slower, or more costly, increases in off-chip bandwidth than in on-chip processing and memory.

The factors enumerated above will render memory bandwidth—particularly pin bandwidth—a more critical and expensive resource than it is today. Given the complex interactions between memory latency and bandwidth, however, it is difficult to determine whether memory-related processor stalls are due to raw memory latency or increased latency from insufficient bandwidth. Current metrics (such as average memory access time) do not address this issue. This paper therefore separates execution time into three categories: processing time (which includes idle time caused by lack of instruction-level parallelism [ILP]), memory latency stall time, and memory bandwidth stall time.

Assuming that a growing percentage of lost cycles are due to insufficient pin bandwidth, the performance of future systems will increasingly be determined by (i) the rate at which the external memory system can supply operands, and (ii) how effectively on-chip memory can retain operands for reuse. By retaining operands, on-chip memory (caches, registers, and other structures) can increase effective pin bandwidth. By measuring the extent to which on-chip memory shields the pins from processor requests, we can determine how much computational power a given package can support.

The miss rate provides a good estimate of traffic reduction for simple caches. Since many techniques can trade increased traffic for decreased latency (i.e., more cache hits), miss rate is not the best measure of traffic reduction for more complex memory hierarchies. The use of *traffic ratios* [18, 20]—the ratio of traffic below a cache to the traffic above it—provides a more accurate measure of how on-chip memories change effective off-chip bandwidth.

Improving the traffic ratio increases the effective off-chip bandwidth, improving performance in systems that stall frequently due to limited pin bandwidth. We propose a new metric, called *traffic inefficiency*, which quantifies the opportunity for reduction in the traffic ratio. We define traffic inefficiency as the ratio of traffic generated by a cache and some optimally-managed memory. This quantity gives an upper bound on the achievable effective bandwidth for a given memory size, package, and program. By decomposing traffic inefficiency into individual components, we can identify where the opportunities lie for improving effective pin bandwidth through traffic reduction.

Section 2 of this paper both defines our execution time decomposition and gives a detailed justification for our claim that latency-tolerance techniques will expose pin bandwidth constraints in future systems. In Section 3, we present measurements that decompose execution time for an aggressive processor and a range of latency-tolerance techniques—showing that bandwidth stalls will indeed be significant for such processors. Section 4 defines traffic ratio and effective pin bandwidth. We then present measurements of traffic ratios for a range of caches, and compute their effective pin bandwidths. Section 5 defines and measures traffic inefficiencies, computes an upper bound on effective pin bandwidth, and uses these results to propose and measure some cache improvements. Finally, Section 6 concludes with a description of possible solutions (both short-term and long-term), related work, and a summary of our main results.

2 Decomposing program execution time

As the performance gap between processors and main memory increases, processors are likely to spend a greater percentage of their time stalled, waiting for operands from memory. The complexity of both modern processors and modern memory hierarchies makes it difficult to identify precisely why a processor is stalling, or what limits its utilization (or IPC).

To understand where the time is spent in a complex processor, we divide execution time into three categories: *processor time*, *latency time*, and *bandwidth time.*[1] Processor time is the time in which the processor is either fully utilized, or is only partially utilized or stalled due to lack of ILP. Latency time is the number of lost cycles due to untolerated, intrinsic memory latencies. By "intrinsic" we mean memory latencies in a contentionless system; latencies that could not be reduced by adding more bandwidth in between levels of the memory hierarchy. Bandwidth time is the number of lost CPU cycles due both to contention in the memory system and to insufficient bandwidth between levels of the hierarchy. This partitioning scheme is superior to using average memory access time, which neither separates raw access latency from bandwidth restrictions, nor translates directly into processor performance (e.g., four simultaneous cache misses in a lockup-free cache will appear as one cache miss latency to the processor, but will be counted as four distinct misses when calculating average memory access time).

Let T_P, T_L, T_B be a partitioning of some program's execution time, T, spent in each of these three categories (processing, latency, and bandwidth, respectively). Let f_P, f_L, f_B be these times normalized to T. Let T_P be the execution time of the program assuming a perfect memory hierarchy (i.e., every memory access completes in one cycle). Let T_I be the execution time of the program assuming an infinitely-wide path in between each level of the memory hierarchy. f_P, f_L, f_B are computed as follows:

$$f_P = T_P / T \tag{1}$$

1. Our decomposition is similar to that used by Kontothannasis et al. to measure cache performance of vector supercomputers [27].

A. Latency reduction	f_P	f_L	f_B
Lockup-free caches	**?**	↓	↑
Intelligent load scheduling	↑	↓	↑
Hardware prefetching	**?**	↓	↑
Software prefetching	↑	↓	↑
Speculative loads	↑	↓	↑
Multithreading	**?**	↓	↑
Larger cache blocks	**?**	↓	↑
B. Processor trends	f_P	f_L	f_B
Faster clock speed	↓	↑	↑
Wider-issue	↓	**?**	↑
Speculative (Multiscalar)	↓	**?**	↑
Multiprocessors/chip	↓	↑	↑
C. Physical trends	f_P	f_L	f_B
Better packaging technology	↑	↓	↓
Larger on-chip memories	↑	↓	↓

Table 1: Estimated effects on execution divisions

$$f_L = T_L / T = (T_I - T_P) / T \tag{2}$$

$$f_B = T_B / T = (T - T_I) / T \tag{3}$$

This characterization of execution time can be converted easily into CPI, if that is the metric of interest. These three categories can be broken down further to isolate individual parts of the system. This enables us to estimate more accurately the performance impact of imperfect components in a complex modern processor—the performance of which cannot be calculated directly from average memory latency and miss rate.

Table 1 presents predictions of how the fraction of time lost to bandwidth stalls will change for future machines. In every row of Tables 1A and 1B, we see that the normalized fraction of bandwidth stalls is increasing. The technological advances listed in Table 1C will mitigate the relative increases of bandwidth-related stalls. Sections 2.1 and 2.2 explain the trends that we present in Tables 1A and 1B. Sections 2.3 and 2.4 describe the physical trends listed in Table 1C. These latter two subsections describe why the physical increases in effective memory bandwidth will be insufficient to satisfy the increased bandwidth needs of future processors.

2.1 Latency-reduction techniques

Improved techniques for reducing and tolerating memory latency can increase f_B—the percentage of execution time spent stalled due to insufficient memory bandwidth. Reduction of memory latency overhead (f_L) aggravates bandwidth requirements for two reasons. First, many of the techniques that reduce latency-related stalls increase the total traffic between main memory and the processor. Second, the reduction of f_L increases the processor bandwidth—the rate at which the processor consumes and produces operands—by reducing total execution time.

The combination of lockup-free caches [28, 40] and careful scheduling of memory operations that are likely to miss [1, 16] is a method of hiding memory latencies. Although this technique does not increase the amount of traffic to main memory, lockup-free caches worsen bandwidth stalls by allowing multiple memory requests to issue—making queueing delays possible in the mem-

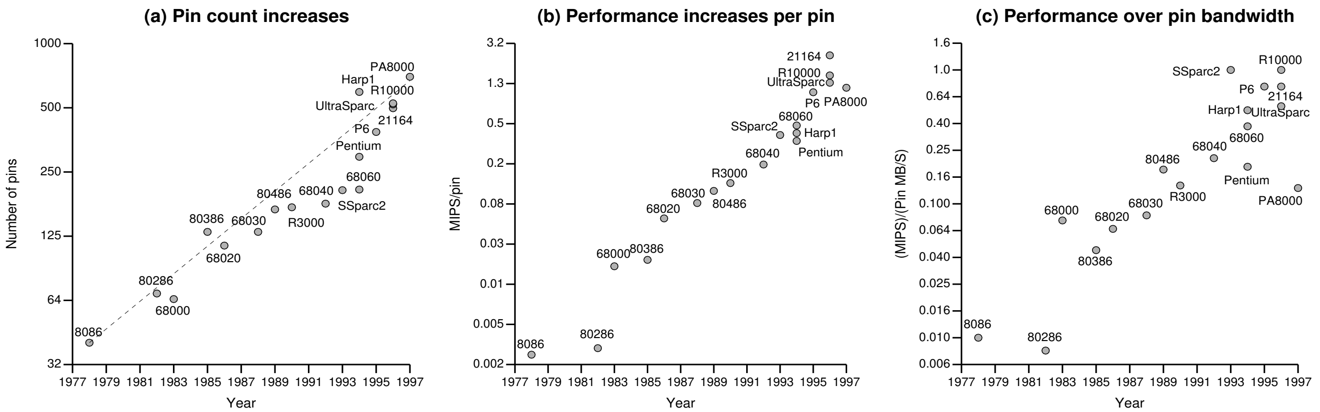

Figure 1. Physical microprocessor trends

ory system. Furthermore, the presence of lockup-free caches will likely encourage more speculative execution.

Both software [6, 8, 26, 32] and hardware [13, 14] prefetching techniques can increase traffic to main memory. They may prefetch data too early, causing other references to evict the prefetched data from the cache before their use. They may also evict needed data from the cache before their use, causing an extra cache miss. Stream buffers [24, 33] prefetch unnecessary data at the end of a stream. They also falsely identify streams, fetching unnecessary data. Speculative prefetching techniques—such as lifting loads above conditional branches [35]—increase memory traffic whenever the speculation is incorrect.

Multithreading increases processor throughput by switching to a different thread when a long-latency operation occurs [30, 38]. Frequent switching of threads will increase interference in the caches and TLB, however, causing an increase in cache misses and total traffic. Poorer cache performance—resulting from the increased size of the threads' combined working set—may offset some or all of the gains of the latency tolerance.

Finally, larger block sizes may decrease cache miss rates. Miss rate improvement occurs until the coarser granularity of address space coverage (i.e., the reduced number of blocks in the cache) overshadows the reduction in misses obtained by fetching larger blocks. Even when larger blocks reduce the miss rate, however, the increased traffic may cause bandwidth stalls that outweigh the miss rate improvements.

2.2 Advanced processors

Several factors other than latency-reduction techniques will increase the needed bandwidth across the processor module boundary. These factors include advanced processor design techniques and shifts in characteristic uniprocessor workloads.

As processors get faster, they consume operands at a higher rate. Faster processor clocks run programs in a shorter time, increasing off-chip bandwidth requirements. Other processor enhancements (such as wider-issue processors) also reduce execution time and increase needed bandwidth.

Processors that rely heavily on coarse-grained speculative execution to increase ILP—such as the Wisconsin Multiscalar [39]—increase memory traffic whenever they must squash a task after an incorrect speculation. Multiple distinct execution units in such processors can execute different parts of the instruction stream simultaneously. This execution may reduce locality in shared, lower-level caches, thus increasing the miss rate, and therefore the total traffic.

The emergence of single-chip multiprocessors would substantially increase the number of data loaded per cycle. The increased bandwidth results primarily from multiple concurrently-running contexts, but also because of shared-cache interference. The primary barrier to the implementation of single-chip multiprocessors will not be transistor availability but off-chip memory bandwidth. If one processor loses performance due to limited pin bandwidth, then multiple processors on a chip will lose far more performance for the same reason.

Finally, throughout the computer industry, there is an increasing software emphasis on visualization, graphics, and multimedia. These codes tend to have large data sets, with much floating-point computation. Traditional caches are remarkably ineffective at reducing the bandwidth requirements of these types of codes [5]. The increased use of this type of software may therefore exacerbate bandwidth limitations.

2.3 Physical limits

The rate of increase of processor pins has traditionally been much slower than that of transistor density. Although large increases in pin counts have recently occurred—and significant breakthroughs in packaging technology undoubtedly lie on the horizon—the issues of reliability, power, and especially cost will prevent pins from sustaining growth in numbers commensurate with the growth rate of processor performance.

Figure 1 shows trends in pin, performance, and off-chip bandwidth from 1978 to 1997. We compiled this data by hand, from both the processors' original manuals and back issues of *Microprocessor Report*. All three y-axes use log scales. The x-axes use a linear scale.

Figure 1a plots the number of pins per processor from 1978 to 1997. We see from the dotted line that pin counts are increasing by about 16% per year. More striking is the result in Figure 1b, which plots processor performance[1] per pin versus time. The raw performance per pin is also increasing explosively, despite the rapid increase in pin count shown in Figure 1a.

Packages and buses are designed to provide sufficient off-chip bandwidth to each generation of processors. Figure 1c—which plots the raw performance-to-package bandwidth ratio over time—shows that performance increases are quickly outstripping the growth in raw peak package bandwidth. The PA-8000 aberration results from that processor's lack of on-chip caches, necessitating an uncharacteristically large package with a high clock rate. Though feasible today from a cost standpoint, this design strategy

1. Performance here is measured in VAX MIPS for the 680x0 and early 80x86 processors, and issue width times clock rate for the others. These two measures cannot be compared directly, but are sufficient to view 20-year trends.

is unlikely to persist very far into the future (as discussed in Section 4.3).

Processors to date have succeeded in keeping a balance between their data requirements and available memory bandwidth. The cumulative effect of the trends and limits described in this section will make this balance increasingly harder to achieve, necessitating changes in the way memory systems are designed. These changes will be especially important when we include the cost of adding sufficient bandwidth to future high-performance processors, since the costs of larger packages grow super-linearly. Cost-sensitive commodity systems will be particularly sensitive to the need for packages that cost too much.

The pin interface is not necessarily the only point in the system where a memory bandwidth bottleneck could arise. Although bandwidth out of commodity DRAMs is presently a concern, high-bandwidth DRAM chips have already appeared on the market (extended data-out, enhanced, synchronous, and Rambus DRAMS [34]). DRAM banks are thus unlikely to become a long-term performance bottleneck. The memory bus is the other possible bottleneck, particularly for bus-based symmetric multiprocessors (SMPs). Widening the bus is a viable solution, as is shifting to a point-to-point network if the bus becomes too great a bottleneck for future SMPs. We believe that among the processor pins, bus, and DRAM interface, continued increases in processor pin bandwidth will be the hardest to sustain.

2.4 On-chip memory increases

Consider a future processor, to be designed as a follow-on to a current processor. Suppose for simplicity that the new processor will have four times as many gates as the current processor. Assume that the area ratio between processor and on-chip memory is unchanged. How will the off-chip bandwidth requirements change for this new chip?

Figure 2 shows the two opposing effects that increasing technology will have upon the balance between f_P and f_B. These graphs are qualitative and do not represent real data. Figure 2a shows the growing gap between processor bandwidth (words consumed per second) and off-chip bandwidth. This trend increases f_B at the expense of f_P.

Figure 2b shows the reduction in off-chip traffic that occurs as on-chip memory size grows per year—enabling greater reuse of operands. For a given program and input, the amount of computation will remain constant, but the off-chip traffic will decrease. This effect produces the opposite effect of the technology curves on Figure 2a—f_P grows at the expense of f_B.

The vertical arrows in the graphs represent the quantity of each trend at a given year. If the arrow marked (1) increases faster than that marked (2), processors will tend to become more memory bandwidth-bound. Conversely, if (2) increases faster than (1), memory limitations will become less of an issue for a given program.

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 $O(N^3)$. Intuitively, then, we might expect the processing requirements eventually to overwhelm the bandwidth limitations, increasing f_P and decreasing f_B.

We performed an analysis similar to Hong and Kung's I/O complexity analysis [21] to show that this argument is misleading. Consider the conventional matrix multiplication, using a tiled algorithm where tiles are of size L, the on-chip memory is of size S, the sides of both matrices are N elements, and $L \ll N$. Previous work showed [21, 29] that the traffic between the on- and off-chip memories is proportional to $2N^3/L + N^2$. Assume that the processor is sufficiently fast for the implemented algorithm to take full

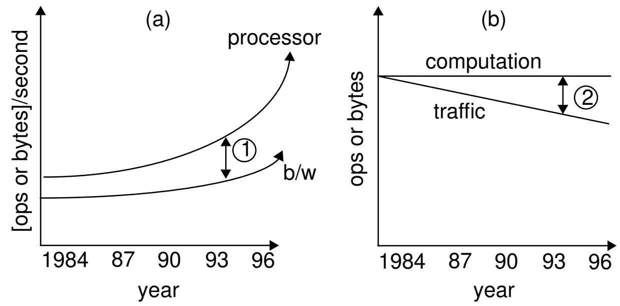

Figure 2. Processing vs. bandwidth changes

Algorithm	Memory	Comp. (C)	Memory traffic (D)	C/D
TMM	$O(N^2)$	$O(N^3)$	$O(N^3/\sqrt{S})$	$\sqrt{k}$
Stencil	$O(N^2)$	$O(N^2)$	$O(N^2/\sqrt{S})$	$\sqrt{k}$
FFT	$O(N)$	$O(N\log_2 N)$	$O(N\log_2 N/\log_2 S)$	$\log_2 k$
Sort	$O(N)$	$O(N\log_2 N)$	$O(N\log_2 N/\log_2 S)$	$\log_2 k$

Table 2: Application growth rates

advantage of the on-chip memory. Holding N constant keeps the amount of computation constant. If the on-chip memory is increased, the program generates less off-chip traffic, allowing the program (assuming a reasonable f_B) to complete in less time. An increase in the on-chip memory by a factor of four would increase L by two, which would reduce the off-chip traffic by nearly half. Therefore, f_B will not decrease so long as the gap marked by (1) also increases by a factor of two.

For the future processor with four times as many gates, the processing speed must increase only by a factor of two (i.e., the square root of the increase in memory size) for the balance between f_B and f_P to remain unchanged. Historically, processor speedup (even ignoring faster technology) has been greater than the square root of the transistor count.

Table 2 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 (on-chip memory size) increases by a factor of k. If this quantity grows slower than the processing speed as S increases, f_P will decline. We believe that such improvement in processing power is attainable, at least for several more generations, and that gap (1) will continue to outpace gap (2) in Figure 2.

3 Measuring execution time decomposition

In this section we show that bandwidth stalls increase as processors and memory hierarchies become more aggressive with latency tolerance. We measure and decompose the execution time of six machines that have a range of latency-tolerance mechanisms in the processor and memory hierarchy.

3.1 Methodology

Our benchmarks consist of seven from the SPEC92 suite [42] and seven from the SPEC95 suite [43]. We selected the benchmarks based on two factors: whether they provided a reasonable range of data set sizes and types of computation, and whether their

Benchmarks SPEC92	Number refs (M)	Data set sizes (MB)	Inputs
Compress	21.9	0.41	1000000 byte file
Dnasa2	181.0	0.18	FFT, MxM=128x64x64
Eqntott	221.1	1.63	`int_pri_3.eqn`
Espresso	22.3	0.04	`mlp4` only
Su2cor	163.4	1.53	`in.short`
Swm	50.6	0.93	180x180, 50 iter.
Tomcatv	104.2	3.67	256x256, 10 iter
SPEC95			
Applu	383.7	32.38	33x33x33 grid, 2 iter.
Hydro2D	263.7	8.71	test data set, 1 iter.
Li	471.3	0.12	`test.lsp`
Perl	1280.8	25.70	`jumble.pl`
Su2cor	533.8	22.53	test data set
Swim	267.4	14.46	test data set
Vortex	1180.3	19.87	test data set

Table 3: Benchmark trace lengths and inputs

simulation times were tractable (or could be made so by reducing input parameters, without skewing the simulation results).

The three integer SPEC92 programs that we used are Compress, Espresso, and Eqntott. The four floating-point-intensive SPEC92 codes are Su2cor, Swm, Tomcatv, and Dnasa2 (two of the Dnasa7 kernels—the two-dimensional FFT and the 4-way unrolled matrix multiply). The three integer SPEC95 codes are Li, Perl, and Vortex. The four floating-point SPEC95 codes are Applu, Hydro2d, Swim, and Su2cor. We present results for both the SPEC92 and SPEC95 versions of Su2cor, and Swm (Swim), since they are different versions with different inputs. Table 3 lists the inputs that we used to generate the traces for each benchmark. It also lists both the number of memory references that we simulated (in millions) and the data set sizes for each benchmark.

We used the SimpleScalar tool set [4] to measure the execution time of simulated processors that use a MIPS-like instruction set. SimpleScalar uses execution-driven simulation to measure execution time accurately. It includes simulation of instruction fetching and system calls. We added a more detailed, multi-level memory hierarchy simulator that includes bus contention. We list the parameters for the simulated memory system in Table 4. We made the memory system slightly more aggressive for the SPEC95 runs by doubling the L2 cache size and splitting the L1 cache into separate instruction and data caches. The bus-to-processor clock frequency ratio is smaller for the SPEC95 runs because we simulate faster processors for SPEC95—the absolute bus speeds are the same or faster for the SPEC95 runs.

To measure f_P, f_L, f_B (derived in Section 2), we execute three simulations for each experiment. To obtain T_P, we run a simulation in which every load and store hits in the L1 cache (one cycle). We measure T_I by simulating a memory hierarchy assuming infinitely-wide paths between adjacent levels of the hierarchy. Finally, we measure T by simulating the full memory system.

The latency-tolerance techniques we evaluate here are the following: increased cache line sizes, the use of lockup-free caches, out-of-order execution with speculative loads, and prefetching. We implemented only one prefetching scheme: tagged prefetch [17]. We assume that our blocking caches can still service hits when they are processing a miss.

Table 5 lists the six experiments (called **A**-**F**) that we ran for each benchmark. Experiments **A**-**C** use an in-order issue, four-way

	SPEC92	SPEC95
L1 cache	128KB unified	64KB I, 64 KB D
	Direct-mapped	
	On-chip, 1-cycle access	
L1/L2 bus	128 bits wide	
	bus/proc clock: 1/3	bus/proc clock: 1/4
L2 cache	1MB	2MB
	4-way set assoc.	
	Off-chip, 30 ns access	
L2/memory bus	64 bits wide	
	bus/proc clock: 1/3	bus/proc clock: 1/4
Memory	90 ns access	
	Infinite banks	

Table 4: Memory system simulation parameters

Experiment	A	B	C	D	E	F
Processor	in-order issue			out-of-order issue		
Branch pred.	8K					16K
Cache	blocking		lockup-free			
L1/L2 blocks	32/64	64/128	32/64			
SPEC92 parameters / SPEC95 parameters						
Speed (MHz)	300/400					300/600
RUU slots	16/64					64/128
L/S Q entries	8/32					32/64

Table 5: Processor simulation parameters

superscalar processor with a two-level branch predictor and two load/store units. Experiments **D**-**F** assume a processor that uses an out-of-order issue mechanism based on the Register Update Unit (RUU) [41], with support for speculative loads. Experiments **D** and **E** are identical except that **E** uses the tagged prefetching scheme (as does experiment **F**).

Table 5 shows how many entries the branch prediction table holds, as well as the cache block sizes, processor speed, number of RUU entries, and the number of entries in the load/store queue. We assumed more aggressive processor parameters for the SPEC95 runs; they are shown in Table 5. Finally, we assume that multiplexed data/address lines are used only on the main memory bus, that all channels are bidirectional, that all memories return the critical word first, and that we have an infinitely-deep write buffer.

3.2 Decomposition results

Figure 3 graphs execution time normalized to the processing time (T_P) of experiment **A**, for each benchmark and experiment. The bars are split into processing cycles, raw latency stall cycles, and limited bandwidth stall cycles. The number atop each bar represents the fractions of the bars that are bandwidth stall cycles.

Experiments **D** and **E** show reductions in f_P due to the out-of-order execution engine. The most aggressive out-of-order processor (**F**) speeds up some benchmarks (Su2cor92, Swm92, Tomcatv) but not others. The SPEC95 benchmarks show little reduction in execution time for **F** because the less-aggressive processors (**A**-**E**) that we used for the SPEC95 runs assume a larger base out-of-order window (64 RUU entries versus 16 for the SPEC92 runs). This larger base window captures much of the available ILP, leaving little additional ILP for experiment **F** to capture.

Using larger block sizes has three effects: increasing both latency and bandwidth stalls (Compress), reducing latency stalls but increasing bandwidth stalls (Su2cor92), or reducing both (Swm92 and Tomcatv). The performance impact correlates directly with the amount of spatial locality that the cache can

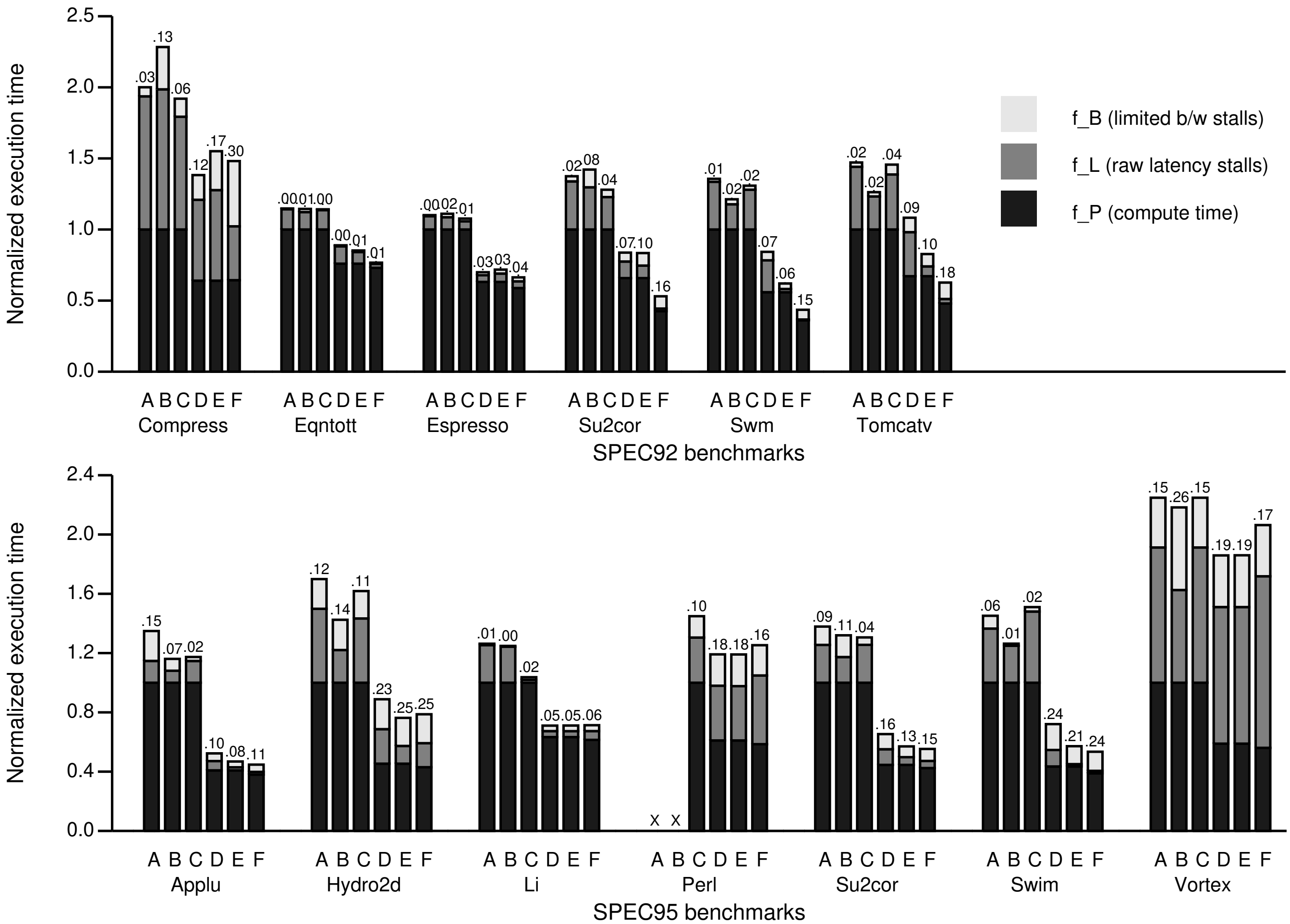

Figure 3. Effect of latency-reduction techniques

Exp.	Compress		Su2cor92		Tomcatv		Applu		Hydro2D		Perl		Swim95		Vortex	
Stall	f_L	f_B	f_L	f_B	f_L	f_B	f_L	f_B	f_L	f_B	f_L	f_B	f_L	f_B	f_L	f_B
A	46.8	3.2	24.6	2.6	30.0	2.1	10.9	15.0	29.4	11.8	---	---	25.2	6.0	40.6	14.9
F	25.6	31.0	3.5	16.3	5.1	18.4	4.0	11.0	20.6	24.8	37.0	16.0	3.1	24.1	56.1	16.7

Table 6: Comparing latency and bandwidth stalls for experiments A and F

exploit for each program. Providing a lockup-free cache (**C**) changes performance very little for all benchmarks; the small reductions in f_L are all nearly offset by corresponding increases in f_B. Larger reductions in f_L are visible when the out-of-order core is added (**D**) to the non-blocking caches.

The most important point that Figure 3 makes, however, supports the thesis of this paper: as the latency-reduction techniques are applied, the bandwidth limitations (f_B) become more severe, generally growing larger than the stalls due to raw latency (f_L). Table 6 shows how the relation between f_L and f_B changes when experiment **F** is compared to experiment **A**. The benchmarks we list here are those that are not cache-bound (Espresso, Eqntott, and Li). In experiment **A**, f_L is greater than f_B for every benchmark except Applu. The relation between latency and bandwidth stalls reverses when we simulate an aggressively latency-tolerant processor. In experiment **F**, f_B is greater than f_L for every benchmark except for Vortex and Perl (and f_B is still significant for both, at 16.7% and 16% of total execution time, respectively).

4 Calculating effective pin bandwidth

Section 3 showed that stalls caused by insufficient memory bandwidth become significant as processors and memory hierarchies attempt to tolerate memory latencies more aggressively. On-chip memory plays a crucial role in reducing off-chip traffic [18]. This reduction increases the effective pin bandwidth, as seen by the processor. When pin bandwidth limits performance, it is important to quantify how much the on-chip memory increases effective pin bandwidth by reducing traffic across the pins.

We therefore measure the *traffic ratio* of a range of caches, which allows us to calculate effective pin bandwidth for a given processor. Hill and Smith proposed using traffic ratios to evaluate the extent to which a cache reduces bus traffic [20]; we generalize their metric to multiple on-chip levels of cache. For a level i in the memory hierarchy, we obtain the data traffic ratio (R_i) by dividing the traffic between levels i and $i+1$ (D_i) by the traffic between levels $i-1$ and i (D_{i-1}):

$$R_i = D_i / D_{i-1} \quad (4)$$

For simple caches with a write-through policy, R_i can be calculated directly from the cache miss rate, the number of issued loads and stores, and the cache block size. A write-back cache decouples the direct correlation between miss rate and traffic ratio. Miss rate becomes a crude approximation of traffic ratios for complicated memory hierarchies: a lockup-free cache may combine two misses with one response from memory, prefetching increases traffic more than it reduces the miss rate, future instruction sets may explicitly move data between levels of the memory hierarchy, and supporting variable transfer sizes makes it difficult to measure cache traffic accurately with miss rate alone.

We use the traffic ratio at each level in the hierarchy to calculate the effective bandwidth to the next lower level of the hierarchy. By dividing the bandwidth from level $i+1$ of the memory hierarchy by R_i, we obtain the *effective bandwidth* from level $i+1$. By taking

$$E_{\text{pin}} = \frac{B_{\text{pin}}}{\prod_{i=1}^{k} R_i} \quad (5)$$

where *k* is the number of levels of on-chip caches, and B_{pin} is the pin bandwidth for the processor in question, we obtain E_{pin}, which is the effective pin bandwidth seen by the processor.

4.1 Simulation methodology

We used trace-driven simulation to measure memory traffic for various cache sizes and configurations. We used QPT to generate traces [19]. The traces contained data memory references but no instructions. QPT handles double-word memory accesses by consecutively issuing the two adjacent single-word addresses.

We used the DineroIII cache simulator [19] to perform our cache simulations. The simulations used the same benchmarks (SPEC92 only) and inputs shown in Table 3. We calculate traffic ratios by running Dinero, and dividing the total traffic by the product of the loads and stores issued and the load/store size. "Total traffic" in this case includes write-back traffic but not request traffic (i.e., addresses). We also flush the cache upon program completion, writing back all dirty data. We include these flushed write-backs in our traffic measurements. Our results contain only data access, not instructions or TLB misses.

4.2 Measured traffic ratios

Table 7 shows traffic ratio measurements for a range of single-level, direct-mapped, 32-byte-block, write-allocate, write-back cache sizes. We saw similar results for caches with higher associativities. The "<<<" symbol indicates that the cache size in question is larger than the benchmark's data set size. We consider this area of the experiment space to be uninteresting, since *R* will always approach 0 when the program runs out of the cache.

When $R_i = 1.0$, a cache generates exactly as much total traffic to memory as would exist with no cache. It is well known [20] that small caches can generate more traffic than a cacheless reference stream; Table 7 demonstrates this result with 1-4KB caches for more than half of our benchmarks. If a block is replaced quickly after its first use—or if there is little spatial locality associated with the access that caused the miss—the other six or seven words loaded with the 32-byte block are superfluous.

Compress and Su2cor generate more traffic with even a 64KB cache than would a cacheless system. Compress repeatedly accesses a hash table, so its memory reference stream contains little spatial locality (a larger block size will consequently waste bandwidth). Su2cor iterates over several large arrays, several of which conflict heavily in its main routine until the cache size reaches 64KB. In contrast to Su2cor, Swm has roughly the same traffic ratio from 16KB to 1MB cache sizes. Swm iterates over large arrays, with a reference pattern that contains little locality and no small working sets [36]. Tomcatv displays similar behavior. In general, R_i ranges between 0.1 and 1.0 for caches that are not overly large or small for a given program.

Since the SPEC92 benchmarks' data sets are not large, these results are conservative—many of these programs run out of the caches and techniques designed to tolerate long latencies have less effect.

The generation of machines that these benchmarks were designed to test did not have on-chip caches larger than 64KB. We therefore calculated the arithmetic mean of the R_i for all caches with sizes greater than or equal to 64KB and less than the data set size of each benchmark. The mean across all benchmarks was 0.51. While this estimate cannot be applied to an individual program/cache combination, it is fair to say that for these benchmarks, reasonably-sized on-chip caches reduce the traffic from the processor by about half.

4.3 Extrapolating pin bandwidth requirements

With our traffic ratios in hand, we now extrapolate pin growth and processor performance, to see what sort of packages we will likely need a decade hence. Figure 1a, shows the rate of growth of processor pins from 1978 to today. We see that the number of pins on processors is increasing at about 16% per year (the dotted line on Figure 1a plots this function).

If we conservatively assume a growth rate of 60% in sustained microprocessor performance—which has been less than the growth rate for the past decade [2]—we can estimate future increases in bandwidth requirements. Assuming that both of these trends persist, and that on-chip traffic ratios remain about the same, we see that in a decade the processor of 2006 will have a package with two or three thousand pins. Even with this large package, the bandwidth requirements *per pin* will be a factor of 25 greater than those of today.

If processors are not to be limited by off-chip bandwidth, at least three possibilities exist (for the processor of 2006):

- Industry may manage to build cost-effective, several-thousand-pin packages clocked at several GHz.
- Industry may instead build a cost-effective package with ten thousand pins and clock it between 0.5 and 1 GHz.
- Improved on-chip traffic ratios increase effective pin bandwidth more than they do today—reducing the need for such huge packages.

The third option listed above is the least costly. To evaluate the potential for package size reduction—given a fixed quantity of on-chip memory—the next section experimentally measures an upper bound on how much effective pin bandwidths may be improved.

5 Improving effective pin bandwidth

We have shown that when a processor employs aggressive latency tolerance, it may spend much time stalled because of limited memory bandwidth. In Section 4, we quantified the amount by which on-chip memory mitigates this performance loss. In this section, we calculate a rough upper bound on effective pin bandwidth by simulating caches that minimize off-chip traffic.

5.1 Traffic inefficiency

To evaluate what percentage of the possible traffic reduction a cache achieves, we measure *traffic inefficiency*—defined as the ratio of traffic produced by the cache in question and the traffic produced by a perfectly-managed cache. We will call this "perfect memory" an ***MTC***, for *minimal-traffic cache*.

Trace	1KB	2KB	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1MB	2MB
Compress	3.03	1.96	1.76	1.59	1.46	1.29	1.10	0.82	0.43	<<<	<<<	<<<
Dnasa2	3.40	2.87	1.34	0.94	0.73	0.62	0.29	0.05	<<<	<<<	<<<	<<<
Eqntott	1.04	0.67	0.55	0.47	0.43	0.39	0.34	0.27	0.18	0.11	0.06	<<<
Espresso	1.43	0.68	0.39	0.20	0.08	0.01	<<<	<<<	<<<	<<<	<<<	<<<
Su2cor	7.44	7.32	6.88	6.11	4.75	2.99	1.43	0.82	0.61	0.29	0.13	<<<
Swm	5.83	5.41	3.94	1.79	0.63	0.60	0.59	0.58	0.58	0.56	<<<	<<<
Tomcatv	2.96	2.91	2.54	1.48	0.87	0.75	0.74	0.73	0.72	0.71	0.33	0.24

Table 7: Traffic ratios for 32-byte block, direct-mapped caches

The traffic inefficiency for level *i* in the memory hierarchy, G_i, is therefore:

$$G_i = \frac{D_{\mathrm{cache}}}{D_{\mathrm{MTC}}} \geq 1 \tag{6}$$

where D_{cache} is the traffic generated by the cache at level *i*, and D_{MTC} is the traffic generated by an **MTC** of the same size and level as the cache.

A memory organization with a $G_i = 1$ is therefore perfectly managed, in terms of memory traffic reduction. Large values of G_i indicate a memory organization that generates much more traffic below it than is necessary.

The traffic inefficiency of a cache allows us to compute an upper bound on effective pin bandwidth. This upper bound is only valid if the processor model remains unchanged; it is possible to change the memory reference stream and therefore further reduce traffic.

Let $\mathrm{OE}_{\mathrm{pin}}$ be the upper bound on effective pin bandwidth. Using traffic ratios and traffic inefficiencies, we can compute this upper bound as follows:

$$\mathrm{OE}_{\mathrm{pin}} = \frac{B_{\mathrm{pin}} \prod_{i=1}^{k} G_i}{\prod_{i=1}^{k} R_i} \tag{7}$$

A simpler expression for this bound is possible using the traffic ratio of the **MTC**, but Equation 7 uses terms for which we present measurements in this paper.

5.2 Measuring a minimal-traffic memory

If we consider only loads from main memory, measuring an **MTC** is straightforward. A cache (discounting stores) generates the minimum possible traffic if it has the following characteristics:

- full associativity,
- the transfer size is equal to the request size,[1]
- it uses an optimal replacement policy, and
- sufficiently low-priority loads can bypass the cache.[2]

If we consider only reads, the optimal replacement policy is Belady's **min** policy [3]. The **min** policy chooses the replacement victim from a set (in this case the entire cache) by evicting the cache block that the processor will reference furthest in the future (or a block that the processor will never reference again).

The **min** policy is not optimal for write-back caches, since there is an additional cost associated with replacing a dirty block. Horwitz et al. proposed an algorithm to manage optimal replacement in the presence of write-backs [22]. We implemented only the **min** algorithm, and not the optimal write-conscious Horwitz algorithm. We believe that the disparity between the two is small, and therefore not worth the additional complexity of simulating the Horwitz algorithm. The traffic inefficiencies presented in the following section are therefore not minimal, but are nevertheless an aggressive bound.

We assumed write-back caches because, for an **MTC**, a write-back policy will always generate less traffic than a write-through policy. The **min** policy will make the write bypass the cache if its line will not be read before it is replaced. We also assumed a write-validate policy [25], in which the cache block is allocated by over-writing it with the store data. This policy will always generate less traffic than allocate-on-write because both the **MTC**'s transfer and address blocks are one word.

We used QPT and Dinero to measure D_{cache} for the cache traffic term of *G*. We wrote our own two-pass simulator, which also used QPT-generated traces, to perform the **MTC** simulation and obtain D_{MTC}. The traffic measurements for both simulators include the same components (e.g., write traffic) as did the traffic ratio experiments.

5.3 Measuring traffic inefficiency

Table 8 shows that there is a wide disparity of values for *G* across the benchmarks. We assumed direct-mapped, 32-byte block caches for these experiments. Four of the benchmarks typically have *G* greater than 20 and less than 100 (Compress, Eqntott, Espresso, and Su2cor)—even for large caches. The other three—Dnasa2, Swm, and Tomcatv—typically have *G* between 2 and 10. These three benchmarks are all scientific codes that display little temporal locality, thus the reference stream contains less opportunity for optimization by a smarter cache. The large jump to a *G* of 124 for Swm with a 1MB cache occurs because the **MTC** (being fully associative) is able contain the entire data set in the cache. Conversely, conflicts between large data structures make caches with associativities of less than four (inclusively) require a size of 4MB to contain the entire data set.

Overall, these numbers demonstrate that there is a significant opportunity to increase effective pin bandwidth, between one and two orders of magnitude, by making better use of the on-chip memory. We now turn to determining which factors contribute to these large gaps. Figure 4 shows a log-log plot of traffic measurements (in KB) versus cache sizes, for three SPEC92 benchmarks. For brevity, we include only Compress, Eqntott, and Swm, since they are somewhat representative of the other benchmarks. The top six lines in each graph represent 4-way, set-associative caches with block sizes from 4B to 128B. The thick dotted line represents a fully-associative, **min**-replacement cache that uses a write-allocate, write-back policy. The thick solid line represents the write-validate, write-back **MTC** that we used for all traffic inefficiency calculations. Large gaps between a line and the **MTC** line indicate large traffic inefficiencies.

There are three factors visible on Figure 4 that contribute to large gaps between cache and **MTC** traffic. The first is increased block size. Compress has little spatial locality, since most of its accesses are to a hash table. Any increase in block size causes a corresponding increase in traffic. The same effect is visible for

1. We assume requests of four-byte words for all experiments.
2. A bypass occurs when a miss has a lower replacement priority than anything else in its set (which is in this case the entire cache).

Trace	1KB	2KB	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1MB	2MB
Compress	25.3	18.4	18.7	19.5	21.9	25.5	29.2	30.7	32.5	<<<	<<<	<<<
Dnasa2	6.2	6.6	6.2	4.7	4.1	4.6	7.0	10.0	<<<	<<<	<<<	<<<
Eqntott	56.3	38.7	34.5	35.8	49.7	94.4	100.5	94.1	72.7	47.7	28.6	<<<
Espresso	18.2	18.8	26.3	40.4	82.2	28.9	<<<	<<<	<<<	<<<	<<<	<<<
Su2cor	14.1	14.5	15.1	16.4	17.2	21.9	20.1	25.7	40.3	28.7	35.8	<<<
swm	22.7	23.4	17.2	7.9	2.8	2.7	2.8	3.0	3.5	5.4	124.1	74.8
Tomcatv	6.4	6.6	6.2	3.9	2.3	2.0	2.0	2.0	2.1	2.4	1.6	3.7

Table 8: Traffic inefficiencies for 32-byte block, direct-mapped caches

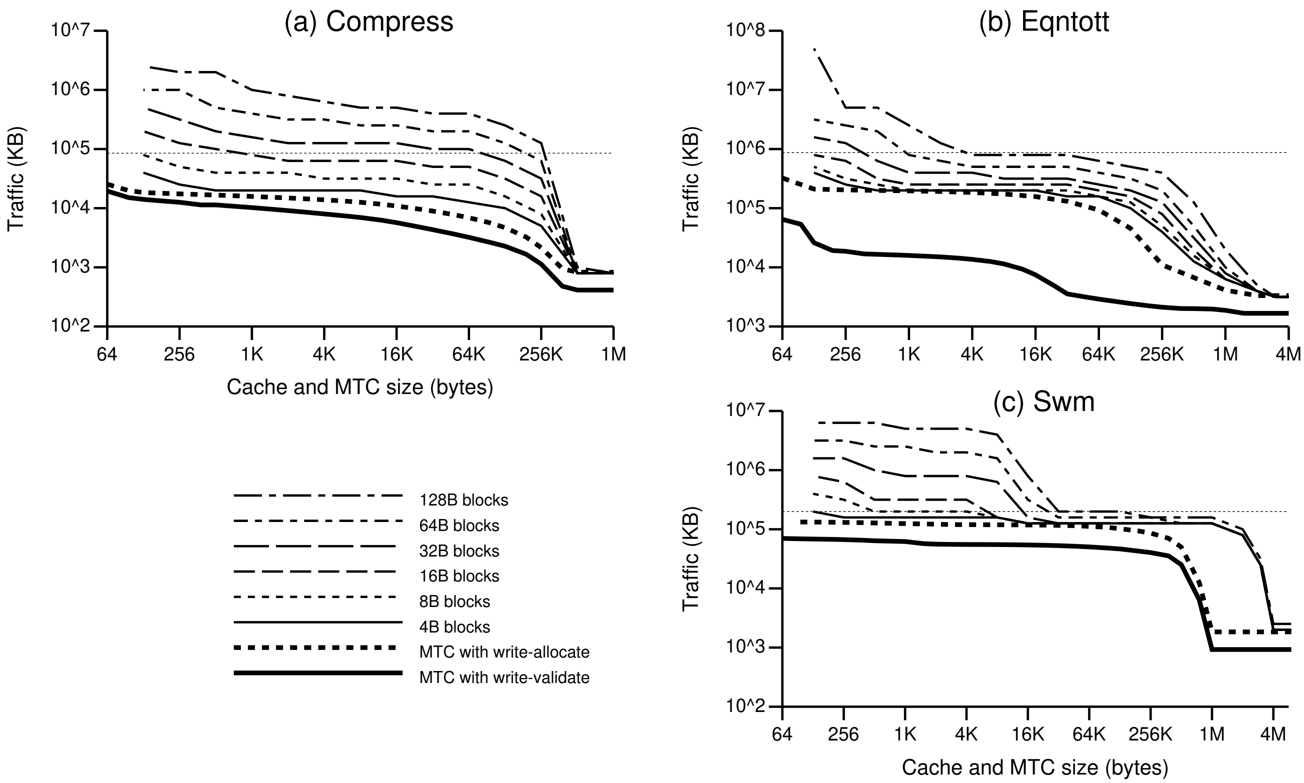

Figure 4. Total traffic generated by different cache and MTC sizes

Eqntott (to a lesser extent), and for Swm when the cache sizes are smaller than about 32KB. Swm shows spatial locality for larger caches because the extra words in larger blocks are used when the block is not quickly replaced—when a small working set fits in the cache. The second factor is associativity, which causes the large gap between caches and the **`MTC`** for Swm at 1MB. The third factor contributing much to the cache/**`MTC`** traffic gap is the write-validate policy, which causes the majority of the gap for Eqntott.

In addition to block size, associativity, and write-validate, there are two factors that enable an **`MTC`** to generate less traffic than caches. These factors are cache bypassing and replacement policy (**`min`** vs. LRU). To better understand which of these factors are significant, we isolate and list four of these factors in Table 9. We did not isolate cache bypassing as a factor; however, Tyson *et al.* recently showed [45] that, for small caches, greater selectivity about what is cached can significantly reduce memory traffic. Table 9 shows how each factor changes G for one cache size per benchmark. We set all cache sizes to 64KB except for Espresso, to which we assigned a cache size of 16KB (because of its small data set). The values in the table show the change in traffic inefficiency as each factor is toggled. Table 10 lists the pairs of experiments (Exp1 and Exp2) run to isolate individual factors. The experiment columns list the replacement policy, set associativity, block size (in bytes), and write policy for each experiment. We measured the traffic effects of block size for both LRU- and **`min`**-replacement caches (experiments III and IV). All experiments (except for Exp2 in experiment V) assumed a write-allocate, write-back policy.

These factors are not independent. Our complete set of results showed that improving one factor tends to diminish the magnitude of another, particularly if the factors are large. What is most significant about Table 9 is the lack of any one factor that dominates the others, across all benchmarks. The factor that makes the largest consistent contribution to traffic reduction, not surprisingly, is reduction of block size. Our results do not consider request traffic, which increases with smaller block sizes, and thus may be biased in favor of smaller blocks.

Using the **`min`** replacement policy has surprisingly small effect. This is because a better replacement policy benefits only codes that have an intermediate amount of locality. Cache replacements occur infrequently for codes that have sufficient locality—reducing the benefits of better replacement policies. Codes that have little temporal locality (such as Swm) hardly benefit from a better replacement policy. These results show that most bench-

Benchmark	Compress	Dnasa7	Eqntott	Espresso	Su2cor	Swm	Tomcatv
Cache size	**64KB**	**64KB**	**64KB**	**16KB**	**64KB**	**64KB**	**64KB**
Associativity	1.8	-3.8	0.5	73.0	8.4	0.1	1.6
Replacement	12.0	8.4	31.0	3.9	4.6	0.3	0
Blocksize (cache)	25.0	2.7	47.0	68.0	14.0	0.3	1.3
Blocksize (MTC)	14.0	0.4	37.0	3.5	5.0	0.3	0.2
Write validate	1.2	1.2	31.0	1.0	1.2	1.3	0.7

Table 9: Inefficiency gap for different optimizations

	Repl. & write policy, assoc., blk. size	
Factor	**Exp1**	**Exp2**
I. Associativity	LRU, 1a, 32B, WA	LRU, fa, 32B, WA
II. Replacement	LRU, fa, 32B, WA	MIN, fa, 32B, WA
III. Blk. size (cache)	LRU, 1a, 32B, WA	LRU, 1a, 4B, WA
IV. Blk. size (MTC)	MIN, fa, 32B, WA	MIN, fa, 4B, WA
V. Write validate	MIN, fa, 4B, WA	MIN, fa, 4B, WV

Table 10: Experimental parameters for Table 9

marks can greatly reduce their total traffic to memory, but require different sets of cache parameters per benchmark to do so.

The wide variance in performance based on block size—for systems which tolerate latency and are at least partially limited by insufficient memory bandwidth—indicates that machines of the future will likely have programmable mechanisms to support variable block sizes. Allowing software-controlled transfer sizes will permit each application to optimize its traffic based on its reference patterns—large transfers to minimize request overhead if there is sufficient spatial locality, and small transfers in the absence of spatial locality. This philosophy can be extended to the other cache parameters, and may become necessary as good use of on-chip memory becomes essential to sustaining reasonable performance.

6 Future solutions and summary

Both limited off-chip bandwidth and growing relative memory access latencies have the potential to seriously degrade program performance. Aggressive latency-tolerance techniques must be implemented with discretion, as they have the potential to worsen performance if memory bandwidth, not untolerated access latencies, is the primary bottleneck for a given program. The potential to overcompensate for latency tolerance will be particularly acute with future processors that rely heavily on speculation to achieve high performance.

A range of techniques for dealing with the growing expense of off-chip accesses exist—above and beyond the brute force, expensive solution of buying more bandwidth to the memory system. We have shown that the potential exists to use on-chip memory much more effectively, greatly reducing the number of requests that must be made off-chip. Not surprisingly, no single technique emerged for making better use of the on-chip memory. This fact suggests that future designers should consider on-chip memory systems that are more flexible, allowing the programmer or compiler to tune the on-chip memory system parameters (such as block size). A more radical extension to the on-chip memory systems is to allow the compiler to manage some data allocation and movement. For example, the kinds of analyses performed for effective register allocation might be readily extended to include other variables that are stored in memory. We are currently investigating both how novel hardware can be controlled by software, and how software might take advantage of this opportunity.

Another short-term solution increasing effective off-chip bandwidth is compression. Researchers have proposed and/or implemented schemes to use compression for data [9], addresses [12], and code [10]. All of these schemes increase effective bandwidth to memory at the expense of some extra hardware on the CPU (and at memory, in the case of the data and address compression).

A more radical technique than compression, which increases effective off-chip bandwidth, is to begin building computational ability into the memory system. The processor would then be able to issue primitives more powerful than simple reads or writes, perhaps even method invocations with the appropriate arguments. The memory system would perform the computation locally and return the result. The idea of "smart memory" is certainly not new, but we may be entering an era when it becomes cost-effective.

A large percentage of today's typical processor chip is already devoted to on-chip memory. When enough transistors are available, a greater capacity on-chip will be more important than having all of the on-chip memory be fast memory. DRAMs may initially appear on multi-chip modules, but will eventually also be incorporated onto the CPU die itself, as new manufacturing processes are developed. We are currently evaluating the design space for mixing DRAM and SRAM on-chip, determining the best way to exploit the extremely high bandwidths attainable from on-chip DRAM banks.

Looking further into the future, we envision a point at which off-chip communication is so expensive that all of the system memory resides on the processor chip (or module). If a system designer wishes to provide more memory than is available on-chip, another of these homogenous, processor/memory modules is added. Off-chip accesses thus simply become communication with another processor, and accesses to remote data have more in common with a page fault than with a cache miss. Whether this point is reached by migrating computational ability into the DRAM system, or by migrating DRAM onto the processor (or both), the end result is the same. Figure 5 shows an example of such a system, in which there is no "dumb" main memory, and cache banks are distributed among the on-chip DRAM banks. We believe that this is how future systems will be designed.

We are currently investigating an execution model for such systems, for both uniprocessor and multiprocessor workloads. Given the limited on-chip memory, multiprocessors are clearly the method of choice for exploiting programs with obvious parallelism. For less-easily parallelized programs, sophisticated hybrid techniques employing some form of data-parallelism, or possibly extensions to ESP as proposed for the Massive Memory Machine [15], might provide competitive performance.

6.1 Related work

A large volume of work on caches appears in the literature, though most has focused on reduction in latency, ignoring the memory bandwidth constraints. Smith's classic survey [37] delineated the fundamental issues concerning caches, including the importance of memory bandwidth. Goodman recognized the importance of a simple memory hierarchy for reducing memory bandwidth, particularly in a multiprocessor environment [18]. Hill

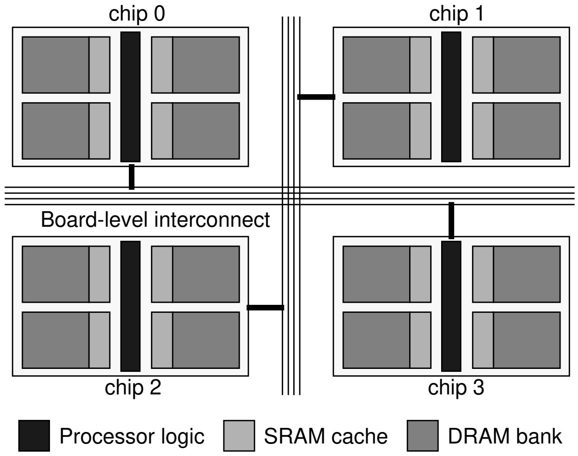

Figure 5. A unified processor/DRAM system

and Smith subsequently measured the trade-offs between miss ratio and traffic ratio by varying block and subblock sizes [20]. McNiven and Davidson looked at reducing traffic between adjacent levels of the memory hierarchy [31]. Sugumar and Abraham developed an efficient method for simulating caches using the **`min`** policy [44]. More recently, Tyson et al. studied ways to bypass the cache with infrequently-referenced data, thereby reducing miss rates [45]. Huang and Shen studied minimal required bandwidths for current processors [23]. They considered only program-generated values, however, without quantifying actual program address behavior.

6.2 Summary

This paper has shown that aggressive implementations of latency-tolerance techniques in future processors will expose memory bandwidth, particularly pin bandwidth, as a severe performance bottleneck. We first surveyed a wide range of these techniques and qualitatively showed that each one exacerbates bandwidth limitations, either directly or indirectly. We also qualitatively analyzed technology trends, showing that future technology is likely to aggravate the bottleneck of the chip boundary. To permit quantification of future bandwidth limitations, we decomposed execution time into processing cycles, raw memory latency stall cycles, and limited bandwidth stall cycles. Using this decomposition, we measured how bandwidth stalls increase, as processors tolerate memory latencies more aggressively. For our applications running on our most aggressive processor, we saw that the stall cycles due to bandwidth exceeded latency stall cycles in all cases but two. Excluding those benchmarks that fit comfortably in the cache, the stall cycles due to bandwidth limitations ranged from 11% to 31% of the programs' total execution time. These measurements have significant implications for the designs of future processors. They also call into question the validity of studies that assume a perfect memory system.

Given the increased importance of pin bandwidth as a precious resource, we introduced the notion of *effective pin bandwidth*—the pin bandwidth as seen by the processor when the on-chip caches are considered. We used *traffic ratios* to compute effective pin bandwidth, and measured these ratios for a range of programs and cache sizes. We found that comparatively large caches eliminated about half of the processor-generated traffic for our small benchmarks. We then introduced the notion of *traffic inefficiency*, which places an upper bound on the amount by which caches can reduce traffic. This bound enabled us to compute the maximal theoretical effective pin bandwidth for a given cache and workload. We measured this bound for a range of programs and caches, showing that effective pin bandwidth could in theory be increased by up to two orders of magnitude—through better management of on-chip memory. We decomposed this gap into individual factors, and used these results to evaluate one and propose several schemes for improving traffic ratios, thereby mitigating pin bandwidth limitations. Finally, we proposed a number of solutions that range from the near-term to the long-term. We hypothesize that all system memory will eventually be coupled with the processor on the die, enabling levels of performance far beyond what we can achieve today.

Acknowledgments

We thank the many people who gave us insightful comments on both this work and this paper: Mark Hill, Guri Sohi, David Wood, Kazuaki Murakami, Alvy Lebeck, Ross Johnson, Mark Callaghan, and Stefanos Kaxiras. We also thank Todd Austin for his valuable assistance with the simulation environment. Finally, we thank the anonymous referees for their extremely detailed and helpful reviews.

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

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

[4] Doug Burger and Todd M. Austin. Evaluating Future Microprocessors: the SimpleScalar Tool Set. Technical Report 1308, Computer Sciences Department, University of Wisconsin, Madison, WI, April 1996.

[5] Douglas C. Burger, Alain Kägi, and James R. Goodman. The Declining Effectiveness of Dynamic Caching for General-Purpose Microprocessors. Technical Report 1261, Computer Sciences Department, University of Wisconsin, Madison, WI, January 1995.

[6] 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.

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

[8] William Y. Chen, Scott A. Mahlke, Pohua P. Chang, and Wen 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.

[9] Daniel Citron and Larry Rudolph. Creating a Wider Bus Using Caching Techniques. In *Proceedings of the First International Symposium on High-Performance Computer Architecture*, pages 90–99, January 1995.

[10] Robert P. Colwell, Robert P. Nix, John J. O'Donnell, David B. Papworth, and Paul K. Rodman. A VLIW Architecture for a Trace Scheduling Compiler. In *Proceedings of the Second Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 180–192, October 1987.

[11] 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, Princeton, NJ, April 1994.

[12] M. Farrens and A. Park. Dynamic Base Register Caching: A Technique for Reducing Address Bus Width. *Proceedings of the 18th Annual International Symposium on Computer Architecture*, 19(3):128–137, May 1991.

[13] 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.

[14] 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.

[15] Hector Garcia-Molina, Richard J. Lipton, and Jacobo Valdes. A Massive Memory Machine. *IEEE Transactions on Computers*, C-33(5):391–399, May 1984.

[16] 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.

[17] J. D. Gindele. Buffer Block Prefetching Method. *IBM Tech. Disclosure Bull.*, 20(2):696–697, July 1977.

[18] 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.

[19] 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.

[20] Mark D. Hill and Alan Jay Smith. Experimental Evaluation of On-Chip Microprocessor Cache Memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 158–166, June 1984.

[21] 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.

[22] 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.

[23] Andrew S. Huang and John P. Shen. A Limit Study of Memory Requirements Using Value Reuse Profiles. In *Proceedings of the 28th International Symposium on Microarchitecture*, pages 71–81, December 1995.

[24] 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.

[25] Norman P. Jouppi. Cache Write Policies and Performance. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 191–201, May 1993.

[26] 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.

[27] L. I. Kontothanassis, R. A. Sugumar, G. J. Faanes, J. E. Smith, and M. L. Scott. Cache Performance in Vector Supercomputers. In *Proceedings of Supercomputing '94*, pages 255–264, November 1994.

[28] 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.

[29] 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.

[30] James Laudon, Anoop Gupta, and Mark Horowitz. Interleaving: A Multithreading Technique Targeting Multiprocessors and Workstations. In *Proceedings of the 6th Symposium on Architectural Support for Programming Languages and Operating Systems*, volume 6, pages 308–318, October 1994.

[31] 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.

[32] 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.

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

[34] Betty Prince. Memory in the fast lane. *IEEE Spectrum*, 31(2):38–41, February 1994.

[35] 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.

[36] Edward Rothberg, Jaswinder Pal Singh, and Anoop Gupta. Working Sets, Cache Sizes, and Node Granularity Issues for Large-Scale Multiprocessors. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 14–25, June 1993.

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

[38] Burton J. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. In *Real-Time Signal Processing IV*, pages 241–248, 1981.

[39] Guri Sohi, Scott E. Breach, and T.N. Vijaykumar. Multiscalar Processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 1995.

[40] Guri Sohi and Manoj Franklin. High-Performance Data Memory Systems for Superscalar Processors. In *Proceedings of the Fourth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 53–62, April 1991.

[41] Gurindar S. Sohi. Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers. *IEEE Transactions on Computers*, 39(3):349–359, March 1990.

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

[43] Standard Performance Evaluation Corporation. *SPEC Newsletter*, Fairfax, Virginia, September 1995.

[44] 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.

[45] Gary Tyson, Matthew Farrens, John Matthews, and Andrew Pleszkun. A New Approach to Cache Management. In *Proceedings of the 28th International Symposium on Microarchitecture*, pages 93–103, December 1995.