LIMITED BANDWIDTH TO AFFECT PROCESSOR DESIGN

Doug Burger

James R. Goodman

Alain Kägi

University of Wisconsin-Madison

An execution-driven simulation measures the time that several SPEC95 benchmarks spend stalled for memory latency, limited-memory bandwidth, and computing.

The phenomenal improvements in microprocessor performance place significant demands on memory systems, requiring a low latency, high-bandwidth stream of operands. Researchers point out that DRAM access latencies (measured in processor cycles) are growing and any request that misses in the caches may eventually take hundreds of cycles to satisfy. These researchers have proposed many techniques to mitigate the penalties of long memory latencies, such as lockup-free caches, cache-conscious load scheduling, hardware and software prefetching, stream buffers, speculative loads and execution, multithreading, data value prediction, and instruction reuse.

Most of these techniques, while reducing the impact of contentionless access latencies, do so at the cost of increasing a program's bandwidth requirements. These latency tolerance (or reduction) techniques may increase a processor's memory bandwidth needs by causing the processor to request the same stream of operands in less time, or by causing the processor to request more data from memory. In turn, these techniques may cause the processor to stall due to queueing in the memory system. (We differentiate between "intrinsic latency" of a contentionless access and latency added by queueing in the memory system, which results from limited bandwidth.)

Neither the long latencies nor the increased bandwidth requirements constitute a "memory wall" that will eventually inhibit improved microprocessor performance. Instead, designers will employ a range of design decisions and new technologies to produce balanced, cost-effective systems. The extent to which long latency and bandwidth requirements affect performance will determine which techniques or technologies are affordable, and/or worth the effort of implementing. Since some of the solutions trade improved latency for increased traffic, or higher bandwidth for increased latency, the relative effects that these two components of memory accesses have on performance is important.

Here, we quantify and compare the performance impacts of memory latencies and finite bandwidth. We show that the implementation of aggressive latency tolerance techniques aggravates stalls due to finite memory bandwidth, which actually become more significant than stalls resulting from uncongested memory latency alone. We expect that memory bandwidth limitations across the processor pins will drive significant architectural change, for the following reasons:

- Continuing progress in processor design will increase the issue rate of instructions. These advances include both architectural innovation (wider issue, speculative execution, and so forth) and circuit advances (faster, denser logic).
- 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.
- Many of the latency tolerance techniques increase the absolute amount of memory traffic by fetching more data than are needed.
- Packaging 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.

These factors will force architectural and system-level change. The range of techniques to improve effective memory bandwidth includes

- wider and/or faster connections to memory,
- larger on-chip caches,
- traffic-efficient requests,
- more efficient on-chip caches,
- logic/DRAM integration, and
- memory-centric architectures.

In an earlier version of this article,[1] we predicted that memory bandwidth, particularly across the processor pins, would drive significant changes in processors and memory systems. This special issue on advanced memory architectures affirms the claims we made. In the ensuing two years, we have seen the popularization of high-bandwidth DRAM interfaces and the beginnings of products integrating memory and logic. Four of the articles in this special issue illustrate this change; two describe high-bandwidth DRAM interfaces (Rambus and SLDRAM), and two others describe logic/DRAM integration (the MSM7680 and the M32R/D).

Decomposing execution time

Without corresponding improvements in the memory system, as the performance gap between processors and main memory increases, the percentage of time a processor spends stalled for memory will also increase. Due to the complexity of modern processors and memory hierarchies, no simple metric (miss rate, for example) sufficiently quantifies the performance impact of an imperfect memory system.

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

Let P, L, and B be the fractions of time that a program spends in each of these three categories (processing, latency stalls, and bandwidth stalls). Let T be the number of cycles in which a given program executes on a realistic system. We can calculate P by measuring the number of cycles it takes a program to execute, assuming a perfect memory hierarchy in which any memory request is returned in one cycle. If T_P is the number of program cycles (assuming a perfect memory), then $P = T_P/T$. We can calculate the stalling time for raw memory latencies by measuring a system with effectively infinite bandwidth, one in which any amount of data may be moved from one level of the memory hierarchy to another in a single cycle. (Requests still incur access latencies to cache and memory banks.) If the time spent to run the program on such a system is T_I, then the number of cycles spent stalling for memory latency is $T_I - T_P$, and $L = (T_I - T_P)/T$. When we add contention (finite bandwidth) back into our measurement of program execution time, the number of cycle increase over the infinite bandwidth case is the number of cycles lost due to finite bandwidth ($T - T_I$). The fraction of time spent stalling due to insufficient bandwidth is therefore $B = (T - T_I)/T$.

Table 1. Estimated effects on execution divisions.

	Effects		
Technique	*P*	*L*	*B*
Latency reduction			
Lockup-free caches	?	↓	↑
Intelligent load scheduling	↑	↓	↑
Hardware prefetching	?	↓	↑
Software prefetching	↑	↓	↑
Speculative loads	↑	↓	↑
Multithreading	?	↓	↑
Larger cache blocks	?	↓	↑
Processor enhancements			
Faster clock	↓	↑	↑
Wider issue	↓	?	↑
Data value speculation	↑	↓	↑
Instruction reuse	↑	↓	?
Speculative threads	↓	?	↑
CMPs	↓	↑	↑

P= processing; L= latency stalls; B= bandwidth stalls; ? = effect is not clear

Table 1 lists latency tolerance optimizations and processor improvements, and shows how we predict P, L, and B will change with each of these improvements. In every row but one, the normalized fraction of bandwidth stalls increases for that optimization.

Latency-reduction techniques. Improved techniques for reducing and tolerating memory latency (reducing L) can increase B—the percentage of execution time spent stalled due to insufficient memory bandwidth. Many of the techniques that reduce latency-related stalls increase the total traffic between main memory and the processor. Furthermore, reducing 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 and careful scheduling of memory operations that are likely to miss effectively hides memory latencies. Although this technique does not increase the amount of traffic to the main memory, lockup-free caches worsen bandwidth stalls by allowing multiple memory requests to issue—making queueing delays possible in the memory system. Furthermore, the presence of lockup-free caches will likely encourage more speculative execution.

Both software and hardware prefetching techniques can increase traffic to main memory. They may prefetch unneeded data, prefetch data that are evicted before use, or may evict other needed data from the cache before use, causing an extra cache miss. Stream buffers prefetch unnecessary data at the end of a stream. They also falsely identify streams, fetching unnecessary data. Speculative fetching techniques—such as lifting loads above conditional branches—increase unnecessary memory traffic when the speculation is incorrect.

Multithreading increases processor throughput by switching to a different thread when a long-latency operation occurs. Frequent switching of threads will increase interference in the caches and the translation look-aside buffer (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 performance gains from improved latency tolerance. Simultaneous multithreading allows multiple instructions from independent threads to be issued simultaneously, increasing contention seen by the individual jobs.

Finally, larger block sizes may decrease cache miss rates. Miss rate reduction occurs until the coarser granularity of address space coverage (the reduced number of blocks in the cache) overcomes 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.

Advanced processors. Improvements to microprocessors other than latency reduction techniques will increase the needed bandwidth across the processor module boundary. As processors get faster, they consume operands at a higher rate. Faster processor clocks will run programs in a shorter time, increasing off-chip bandwidth requirements (and therefore decreasing P). Wider-issue processors that exploit more ILP similarly reduce execution time and increase needed bandwidth.

Speculation can also increase bandwidth requirements. Data value speculation, when successful, allows the computation to proceed more quickly (increasing the needed rate of operands, but not the total number of operands, as the data must still be computed or loaded from memory to verify the speculation). Instruction reuse, conversely, replaces computation and memory accesses with on-chip table lookups and is thus the only entry listed in Table 1 that reduces B. Future microprocessors that rely on coarse-grained speculative threads to improve ILP—such as multiscalar processors[2]—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 increase the number of data loaded per cycle in a manner similar to multiscalar processors. The increased bandwidth requirements result primarily from multiple, concurrently running contexts or threads, but they will also increase 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, multiple on-chip processors will lose far more performance for the same reason. (If the multiple threads share most or all of their working sets, however, the off-chip bandwidth will be less of an issue.)

Since the majority of microprocessor enhancements discussed here aggravate bandwidth limitations (if they exist), designers must use other solutions to mitigate the performance impact of limited bandwidth.

Execution time decomposition

We hypothesized that bandwidth stalls increase as processors and memory hierarchies become more aggressive with latency tolerance. To test our hypothesis, we simulated the execution time of several systems using seven benchmarks from the SPEC95 suite. We used four integer benchmarks (compress, vortex, ijpeg, and perl), and three floating-point benchmarks (swim, su2cor, and tomcatv). We ran each benchmark using the test input set from SPEC, simulating them until completion. (We used the "train" input set for ijpeg and compress and reduced the number of iterations for vortex.) We used the SimpleScalar tool set to measure the execution time of each system with execution-driven simulation. SimpleScalar contains a detailed, out-of-order, speculative processor simulator, which executes binaries compiled to the SimpleScalar instruction set (similar to that of MIPS). We modified the memory system to simulate lockup-free caches, accurate bus contention, and address translation.

Three simulations for each benchmark measured the P, L, and B components of execution time described earlier. To measure P, we simulated a processor for which every load and store completes in one cycle. We measured L by simulating a memory hierarchy in which all buses are sufficiently wide that all transmissions complete in one bus cycle. Finally, we measured B by simulating the full memory system, accounting for contention.

We simulated six different processor configurations for each benchmark to evaluate the effect of various latency tolerance techniques. The latency tolerance techniques we evaluated are 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.[3] We assumed that our blocking caches can still service hits while they are processing a miss. Table 2 (next page) lists the specific experiments (labeled A-F) that we ran for each benchmark. Experiments A, B, and C use an in-order issue, four-way superscalar processor with a two-level adaptive gshare branch predictor and two load/store units. B assumes cache blocks that are twice as big at each level as those in experiment A (keeping total cache size fixed). Experiment C is the same as A except that it uses a lockup-free cache, as do D-F.

Experiments D-F assume a four-way, out-of-order issue processor based on the register update unit (RUU) structure,[4] which allows loads to issue speculatively. Experiments D, E, and F are identical except that E and F use tagged prefetching, and F uses a more aggressive processor core than do A-

Table 2. Processor simulation parameters.

Parameter	Experiment A	B	C	D	E	F
Processor	In-order issue			Out-of-order issue		
Clock speed	500 MHz					1 GHz
RUU slots	128					256
L/S Q entries	64					128
Branch predictor	16K					32K
Cache	Blocking		Lockup-free			
L1/L2 block sizes	32/64	64/128	32/64			
HW prefetch	No				Yes	

E. Table 2 also lists the processor parameters for each experiment (clock speed, size of branch prediction table, load/store queue, RUU entries, and so forth.)

Table 3 lists the parameters for the memory hierarchy. In addition to those parameters, we assumed that multiplexed data/address lines are used only on the main memory bus, that all channels are bidirectional, and that all memories return the critical word first.

Figure 1 displays the execution times for each experiment. The execution times for each benchmark are normalized to the perfect memory system experiment of experiment A.

Our results show that for simple processors with unaggressive memory systems (experiment A) the benchmarks we measured spend an (arithmetic) mean of 28% of their cycles stalled for memory. More aggressive modern processors (experiment F), which incorporate many of the latency-tolerance features previously discussed in this article, spend fully 50% of their cycles stalled for memory in the simulated benchmarks. The aggressive processors spend more of their time stalled for limited bandwidth; a mean 29% of total cycles in experiment F, as opposed to a mean of 15% in experiment A. Limited bandwidth accounts for the preponderance of the memory stalls for four of the benchmarks in experiment F, but latency stalls still represent a large fraction of memory stalls for perl, tomcatv, and vortex. This effect is due to poor instruction cache performance for these benchmarks. The latency stalls increase (for perl, tomcatv, and vortex) when going from experiment E to F because the faster clock in experiment F makes instruction cache misses more expensive; although the interconnect speed is scaled proportionally to the faster clock, the absolute memory bank access times are not changed. Ijpeg differs from the rest of the benchmarks in that it spends little time stalled for memory. Its reference stream has excellent locality, so both the data and instruction cache miss rates are low (1% for the level-one data cache, and close to 0% for the level-one instruction cache and level-two unified cache). For the other benchmarks, however, our results show that memory bandwidth stalling becomes a major, and sometimes dominant, component of total execution time.

Future solutions

Both limited off-chip bandwidth and growing relative memory access latencies have the potential to degrade program performance seriously. Our results show that in many cases, 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, which will rely heavily on speculation to achieve high performance.

Table 4's analysis (see p. 60) is similar to that of Table 1 for a range of solutions that increase actual or effective memory bandwidth. High-bandwidth DRAMs increase bandwidth but may increase per-request latency as well. Larger caches improve average access latency and reduce traffic. Traffic-

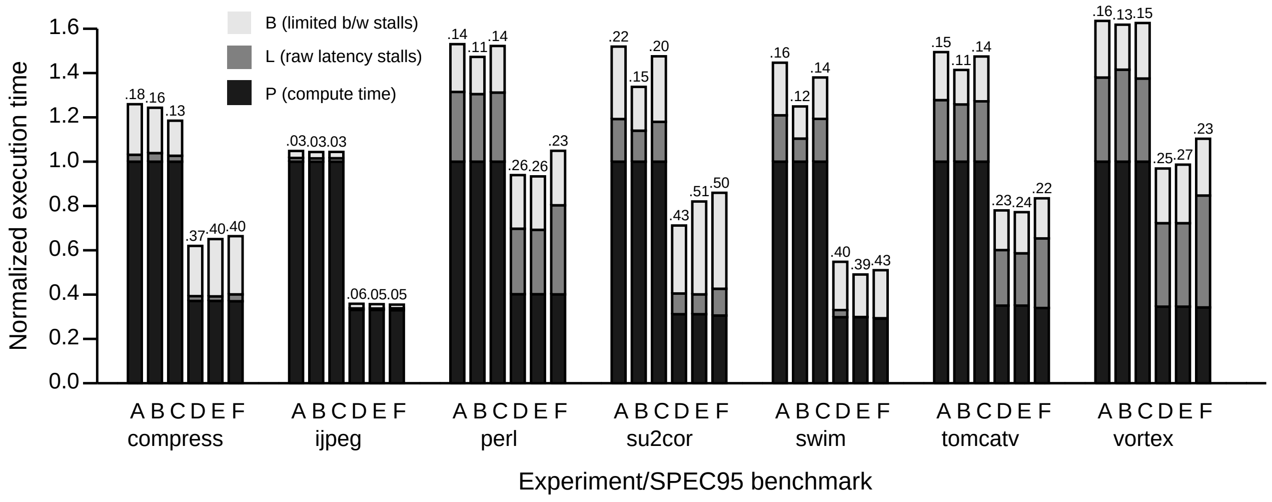


Figure 1. Performance breakdown of execution time. The number atop each bar represents *B*.

Table 3. Memory system simulation parameters.

Structure	Parameters
L1 cache	64 Kbytes I, 64 Kbytes D, on-chip, 1-cycle access
L1/L2 bus	128 bits wide, bus/processor clock: 1/5
L2 cache	2Mbytes, 4-way set associativity, off-chip, 14-ns access
L2/memory bus	64 bits wide, bus/processor clock: 1/5
Memory	80-ns access, no bank conflicts

efficient requests are the analog to latency-tolerance techniques; they will reduce traffic but may exacerbate *L* when not used effectively (thus they may increase or decrease *P*). More efficient on-chip caches will reduce *B* but may increase *L* if the overhead of maintaining more efficient cache use (increased hit latency) outweighs the benefit (reduction in misses). Logic/DRAM integration will increase *P*, as do larger caches. Finally, memory-centric architectures will certainly lend themselves to improving memory system performance, but they are hampered by the need for an implementation acceptance path.

High-bandwidth DRAM interfaces. The first, most obvious solution to limited memory bandwidth is to provide a higher-bandwidth memory system (effectively buying more bandwidth). Two of the articles in this special issue deal with high-bandwidth DRAM interfaces: Rambus (RDRAM) and synchronous link (SLDRAM), which are expected to provide enough memory bandwidth for the next generation of high-performance microprocessors. The key question is not whether providing enough memory bandwidth with each generation is possible, but whether providing enough for each generation is cost-effective. In particular, providing enough pins at a low-enough cost will be a significant challenge in the near future.

The rate of increase of processor pins has traditionally been much slower than that of transistor count. Although large increases in pin counts have recently occurred—and 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. (This statement is especially true since the costs of a package grow superlinearly with the number of pins.) Figure 2 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 2a plots the number of pins per processor. We see from the dashed line that pin counts are increasing by about 16% per year. More striking is the result in Figure 2b, which plots processor performance per pin versus time. The raw performance per pin is also increasing explosively, despite the rapid increase in pin count shown in Figure 2a. Packages and buses are designed to provide sufficient off-chip band-

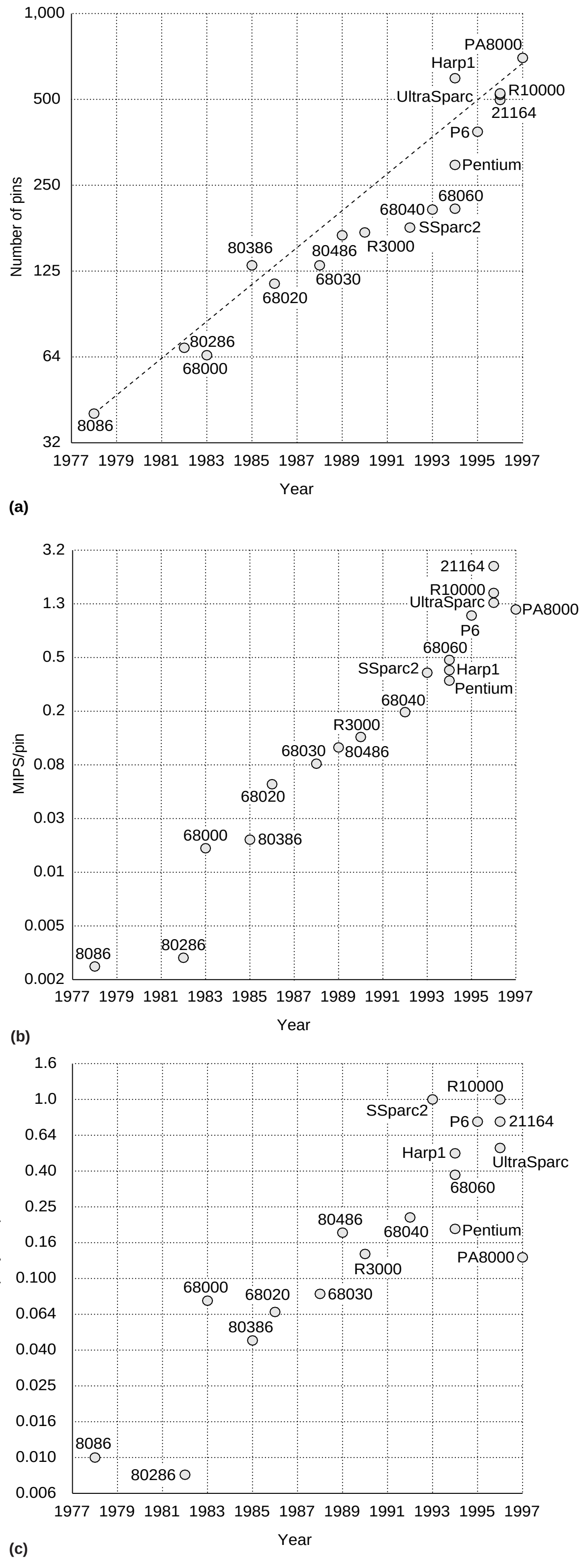


Figure 2. Physical microprocessor trends: pin count increases (a), performance increases per pin (b), and performance over pin bandwidth (c).

Table 4. Effects of memory bandwidth solutions.

	Effect		
Solution	*P*	*L*	*B*
High-bandwidth DRAMs	↑	?	↓
Larger on-chip caches	↑	↓	↓
Bandwidth-efficient requests	?	↑	↓
More efficient caches	↑	?	↓
Logic/DRAM integration	↑	↓	↓
Memory-centric architectures	↑	↓	↓

width to each generation of processors. Figure 2c—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.

In the earlier version of this article,[1] we measured traffic ratios for a range of cache configurations using the SPEC92 benchmarks. The traffic ratio is the number of bytes of bus traffic generated by cache misses divided by the number of bytes of traffic with no cache. While the traffic ratio depends on cache size, the mean traffic ratio of all our experiments was 0.51. Using this value as a rough estimate of future traffic ratios, we can extrapolate pin growth and processor performance to see what sort of packages we will likely need by 2007.

If we assume an annual growth rate of 60% in sustained microprocessor performance—which has been less than the growth rate for the past decade—we can estimate future increases in bandwidth requirements. Assuming 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 must build cost-effective, several-thousand-pin packages clocked at several GHz.
- Industry must build a cost-effective package with 10,000 pins and clock it between 0.5 and 1 GHz.
- Other techniques may improve the effective pin bandwidth more than they do today—reducing the need for such huge packages.

We now turn to a discussion of how the effective pin bandwidth may be increased without simply paying for more raw bandwidth, thus mitigating the high costs of future packages.

Larger on-chip caches. Larger caches improve effective bandwidth by sending fewer requests (misses) across the interconnect. Growing on-chip caches will soon reach the multi-megabyte range. Caches currently account for 50% to 80% of the transistor budget of microprocessors, and this fraction is increasing. When 100 million- and eventually billion-transistor processors become available, they will likely contain tens or hundreds of megabytes of on-chip cache memory. Although caches this large will be able to hold the working sets for many applications, there will always be programs whose access patterns aren't amenable to caching. There will also be programs for which their working sets are too big to fit in even those gargantuan caches.

Even today, some designers are consciously using large on-chip caches to increase their effective pin bandwidth, cutting down on the actual bandwidth needed across the processor package. Current Hewlett-Packard's PA series of microprocessors do not contain on-chip caches or only extremely small caches—the primary cache is off-chip. This organization required a large, high-performance package (1,085 pins in the PA-8200). However, the recently announced PA-8500 reverses this philosophy dramatically; the cache organization includes 1.5 Mbytes of on-chip cache, thereby reducing the package size (which contains a "mere" 544 pins). Not only are the power and cost reduced over that of a larger package, but the PA-8500 saves die area by having fewer pins, thus reducing the total pad and driver circuitry.

Traffic-efficient requests. Effective bandwidth may be increased not only by sending fewer requests but also by increasing the information content or utility of those bytes that are sent. By requesting smaller blocks, a processor gets less implicit prefetching (thereby increasing *L*) but sends fewer unneeded bytes across the interconnect, thus reducing *B*. Our previous paper[1] measured the traffic generated by a "minimal traffic cache," which was organized to minimize bus traffic (small blocks, optimal replacement, full associativity, and a write-validate, write-back policy). Our results showed that, of these factors, small blocks were by far the largest contributing factor to reducing miss traffic (small blocks accounted for 42% of the large differential, averaged over all the benchmarks). Sending smaller blocks, however, increases transmitted tag and arbitration overheads (buses are generally much better at burst transfers); these factors must be weighed against the benefit of sending smaller blocks.

One implementation that could conceivably handle smaller requests without exploding the miss rate is a sector cache with small subblocks, in which either a subblock or the entire address block could be loaded on a miss, depending on the expected block access pattern.

Another method for improving the effective bandwidth of an interconnect is compression. Researchers have proposed and/or implemented schemes to use compression for data,[5] addresses,[6] and code.[7] All of these schemes increase effective bandwidth to memory at the expense of some extra hardware on the CPU (and for memory, in the case of the data and address compression). Current technology trends (computation growing cheaper relative to expensive communication) will make compression more attractive in the future.

More efficient on-chip caches. Although caches are quite effective at capturing temporal and spatial locality in the dynamic reference stream, they do so in a crude fashion, and are far from optimal in their use of buffering active data. Both our group and Wood et al.[8] found that the fraction of data in caches that is live (that is, will be referenced again before it is evicted from the cache) is quite small—typically between one twentieth and one third. The implication of this result is that caches have the potential to be much better managed. Our results confirm this hypothesis; a cache that is optimally managed (in terms of minimizing traffic) produced between two and 100 times less traffic. Huang and Shen confirmed this result in their study of minimal required bandwidths for current-generation processors.[9] They found that if an on-chip memory has no naming or mapping restrictions, the available on-chip buffering and off-chip bandwidth are more than sufficient. The question, of course, is how to extract some of this capability while still permitting a feasible implementation.

One way to improve efficiency in the cache is to cache objects at a finer granularity than a cache line when they are unlikely to exhibit spatial locality. Small blocks increase tag overhead, and will perform poorly when larger objects have spatial locality. Seznec's decoupled sector cache allows a finer grain of access, while preventing space in the unused lines of a given sector from being wasted (by storing lines from other sectors therein).[10]

Conflicts in the cache may remove lines before they are used, thus effectively "killing" the line. Seznec showed how to reduce both the number and variability of conflicts with skewed-associative caches. Tyson et al. proposed a bypassing scheme[11] to address capacity misses in the cache, by statically and/or dynamically determining which data have little temporal locality, and then not loading those data into the cache upon a miss.

Logic/DRAM integration. Putting the processor on the same die (or in the same package) as main memory could eliminate the need for expensive, high-bandwidth interchip interconnects. Industry is already taking some steps in that direction for embedded processors and graphics controllers, as we can see from the two articles in this special issue on the MSM7680 and the M32R/D. Academic research has also been working in this area; our group and Berkeley's IRAM group have been looking at issues related to processor/memory integration.

Logic and DRAM manufacturing processes, however, are fundamentally different. DRAM processes typically support multiple layers of polysilicon, few metal layers, and three-dimensional structures for maximizing capacitor area. Logic processes tend to have many more metal levels and are optimized for transistor-switching speed. DRAM cells in a logic process would not be particularly dense (estimates vary from four to 20 times less dense than in an optimized DRAM process). Gates in a DRAM process are much slower than their logic process counterparts. Furthermore, the two types of processes are diverging.

The benefits of having denser on-processor memory cells, or faster on-DRAM gates, however, may drive the development of hybrid processes that—even though they don't match their optimized counterparts for density or speed—nevertheless improve system performance. In addition, the average number of memory chips in lower-end systems (PCs and workstations) is dropping, and may decrease to one in the next decade. On-processor storage capacities are also converging with average main memory sizes (for personal computers), albeit slowly. Given those trends, it is conceivable that all system memory may someday reside on a single processor chip or module (for personal computers, at least). The question of expansibility then comes into play—how to expand the system memory (add more of the integrated chips, or just reintroduce "dumb RAM"?) If more integrated chips are added, how should programs be run?

Designers must ensure the processor and supporting memory systems are in balance.

Memory-centric architectures. If processing becomes cheap enough, and/or communication becomes too expensive (either across pins or with increased intrachip wire delays), it is likely that designers will put functional units wherever there is storage—we call them memory-centric architectures. The challenges in such architectures are distributing the data effectively and orchestrating the flow of control.

For certain regular, fine-grained, single-instruction, multiple-data (SIMD) types of codes, the processing-in-memory approach, in which small processors are embedded in memory arrays, can show large performance improvements. This is the case with Computational RAM (CRam) and processor-in-memory (PIM) research. The application base of this approach is limited, however.

We have proposed and evaluated a memory-centric architecture called DataScalar, which targets scalar, hard-to-parallelize codes.[12] Processors are coupled with regions of the main memory, and each processor runs the program redundantly, broadcasting operands from its local memory to the other processors. If a processor needs an operand from a remote memory bank, it does not need to request that operand. It simply waits for the operand to arrive, since it will be sent by the processor at the remote memory.

We are currently extending this model to allow processors to select dynamically a computational task for which all the operands are local, execute that code alone, and send the updated state (bundled register values) to the other nodes. We are also implementing support for fine-grained data migration, aggressive speculation, and communication of control decisions, to achieve performance gains above the DataScalar base model. This architecture is a good match for a world in which limited memory bandwidth, wire delays, and off-chip communication delays dominate performance.

FOR A COMPUTER SYSTEM to exhibit good cost/performance, designers must ensure the processor and supporting memory system are in balance. The growing cost of communication—manifested as the growing number of cycles required for main memory accesses—has caused designers to implement latency-tolerance techniques such as prefetching, nonblocking caches, and out-of-order execution. None of these techniques mitigate memory or pin bandwidth limitations, and in many cases they aggravate the performance losses by increasing the off-chip contention.

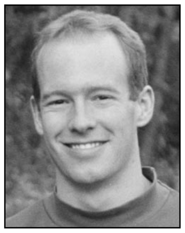
Paying the high cost of a fast or wide path to DRAM is likely to be part of the solution, but many other techniques can reduce bandwidth requirements. Designers can implement a less-expensive main memory system and interface that nevertheless provides the processor with a sufficient rate of operands. Large on-chip caches can reduce the number of off-chip accesses that must be made. Smarter cache management may reduce off-chip traffic even further. Moving the main memory closer to the processor (or moving part of it on the processor) will even further reduce the frequency with which requests must go off chip. Finally, farther in the future, new memory-centric architectures may be developed specifically to improve the memory system's cost and performance. It is likely that most of these solutions will eventually be used synergistically to provide a sufficiently high-bandwidth memory system at an acceptable cost.

Acknowledgments

We thank Ken Sakamura for including this revised work in his special issue, and Steve Diamond and Marie English at *IEEE Micro* for giving us the opportunity to submit our work. We also thank our funding sources—the National Science Foundation, the Intel Research Council, and Sun Microsystems—for the generous support that made this work possible.

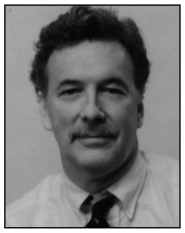
References

1. D. Burger, J.R. Goodman, and A. Kägi, "Memory Bandwidth Limitations of Future Microprocessors," *Proc. 23rd Ann. Int'l Symp. Computer Architecture*, Assoc. of Computing Machinery, New York,1996, pp. 79-90.
2. G.S. Sohi, S.E. Breach, and T.N. Vijaykumar, "Multiscalar Processors," *Proc. 22nd Ann. Int'l Symp. Computer Architecture*, ACM, 1995, pp. 414-425.
3. J.D. Gindele, "Buffer Block Prefetching Method," *IBM Tech. Disclosure Bull.*, Vol. 20, No. 2, 1977, pp. 696-697.
4. G.S. Sohi, "Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers," *IEEE Trans. Computers*, Vol. 39, No. 3, 1990, pp. 349-359.
5. D. Citron and L. Rudolph, "Creating a Wider Bus Using Caching Techniques," *Proc. First Int'l Symp. High-Performance Computer Architecture*, IEEE Computer Society Press, Los Alamitos, Calif., 1995, pp. 90-99.
6. M. Farrens and A. Park, "Dynamic Base Register Caching: A Technique for Reducing Address Bus Width," *Proc. 18th Ann. Int'l Symp. Computer Architecture*, ACM, 1991, pp. 128-137.
7. R.P. Colwell et al., "A VLIW Architecture for a Trace Scheduling Compiler," *Proc. Second Symp. Architectural Support for Programming Languages and Operating Systems*, ACM, 1987, pp. 180-192.
8. D.A. Wood, M. D. Hill, and R. E. Kessler, "A Model for Estimating Trace-Sample Miss Ratios," Tech. Report TR-1000, Computer Sciences Dept., Univ. of Wisconsin, Madison, Wisc., 1991.
9. A.S. Huang and J.P. Shen, "A Limit Study of Memory Requirements Using Value Reuse Profiles," *Proc. 28th Int'l Symp. Microarchitecture*, IEEE CS Press, 1995, pp. 71-81.
10. A. Seznec, "Decoupled Sectored Caches: Conciliating Low Tag Implementation Cost and Low Miss Ratio," *Proc. 21st Ann. Int'l Symp. Computer Architecture*, ACM, 1994, pp. 384-393.
11. G. Tyson et al., "A Modified Approach to Data Cache Management," *Proc. 28th Int'l Symp. Microarchitecture*, IEEE CS Press, 1995, pp. 93-103.
12. D. Burger, S. Kaxiras, and J.R. Goodman, "DataScalar Architectures," *Proc. 24th Ann. Int'l Symp. Computer Architecture*, ACM, 1997, pp. 338-349.



Doug Burger is a PhD candidate at the University of Wisconsin-Madison. His dissertation topic is memory hierarchies for advanced microprocessors.

Burger received an MS from the University of Wisconsin-Madison and a BS from Yale University, both in computer science. He is an Intel Foundation Graduate Fellow and a student member of the IEEE, the Computer Society, and the ACM.



James R. Goodman is professor and chair of computer sciences at the University of Wisconsin-Madison. His current research focuses on high-performance memory systems and computer systems of the future.

Goodman received a PhD from the University of California at Berkeley. An early contributor to multiprocessor snooping cache literature, he has actively participated in the development of IEEE Std 896 (Futurebus) and 1596 (Scalable Coherent Interface). He has published papers in the areas of cache-coherence algorithms, shared-memory multiprocessor architectures, database systems, interconnection networks, virtual memory, memory-register organization, and memory systems design.

Alain Kägi is a PhD candidate at the University of Wisconsin-Madison. His dissertation research concerns efficient synchronization primitives for shared-memory multiprocessors.

Kägi received an MS from the University of Wisconsin-Madison and a computer science diploma from the Swiss Federal Institute of Technology (ETH) in Zürich.

Send correspondence about this article to Doug Burger, University of Wisconsin-Madison, Computer Sciences Dept., 1210 West Dayton St., Madison, WI 53706; dburger@cs.wisc.edu.