DataScalar Architectures and the SPSD Execution Model

Doug Burger, Stefanos Kaxiras, and James R. Goodman

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[1]

The increasing power of commodity microprocessors is forcing system designers to provide more complex and expensive memory hierarchies. A potentially cheaper and better-performing alternative in the long run is to integrate the processor and main memory on the same die or module. In this paper, we propose an architecture (DATASCALAR) and an execution model (SPSD) that permit efficient execution of uniprocessor programs across multiple integrated processor/memory modules. We then describe four features of this proposal that permit improved performance: ESP gains*,* memory prefetching*,* result communication*, and* hybrid parallel execution*. Finally, we present examples and measurements, which give evidence that each feature will improve performance on future systems that have very expensive off-chip communication.*

1 Introduction

Modern microprocessors exhibit very high levels of performance, which nevertheless continue to increase exponentially [1]. These CPUs require a memory system that can provide operands both very quickly and at a very high rate. Current memory systems therefore employ a deep hierarchy with complex features, to provide high performance at acceptable cost. These memory systems generally have split level one instruction and data caches, and may have two levels of cache on the CPU die itself [2]. In addition to having two or three levels of caches of different sizes, with differing line sizes and associativities, these caches are generally lockup-free [20], allowing multiple misses to be outstanding at all levels.

Even with such aggressive memory hierarchies, however, modern processors spend much of their time stalling for needed operands, both instructions and data [4]. This situation is unlikely to be rectified, given the continuing increases in processor performance, increases of main memory sizes, and increases in the disparity between processor and memory cycle times [5]. Memory latency tolerance/reduction techniques—such as non-blocking caches [20, 11], hardware and software prefetching [6, 8, 7, 14, 19, 22], multithreading [21, 27], and out-of-order execution [32, 30]—may reduce memory-related processor stalls until available memory bandwidth is saturated. A recent study showed that aggressively latency-tolerant processors owe fully half of their memory stall times to limited off-chip bandwidth [4].

Although scaling up cache size, levels of cache, and cache complexity *ad infinitum* may be one way to keep the memory system balanced with processor performance, tightly coupling the processor with main memory may eventually prove to be a more cost-effective solution. Integrating main memory onto the same multi-chip module (and perhaps eventually onto the CPU die itself), serves to reduce both high memory latency and memory bandwidth limitations. Many of the other techniques, conversely, trade memory access latency off for memory bandwidth, never reducing both simultaneously.

Current trends in processor design indicate that more and more of the chip will be devoted to memory. Indeed, a huge portion of a modern microprocessor chip is dedicated to the top of the memory hierarchy: the registers and one or two levels of cache. Assuming this trend continues, and as the gap between on-chip and off-chip memory latency grows, it may become desirable to integrate the entire memory on-chip, leaving off-chip accesses for references that are treated more like page faults than cache misses. The on-chip memory may even include high-density memory, such as DRAM. Such processor/memory integration may permit a less-complex, more cost-effective memory hierarchy of equivalent performance to the current model, though it will impose rigid constraints on memory size. Extending the memory system becomes problematic. We partition programs for such systems into three categories:

- The program's data set fits in on-chip DRAM, not requiring any external memory other than disk or other long-latency memory.
- The program's data set is larger than the on-chip memory, but a "dumb" off-chip memory system (similar to those of today) can support reasonable processor performance,
- The program's working set is so large that the processor spends much of its time waiting for off-chip accesses.

This paper is targeted at the third category, proposing a system to execute a conventional single-processor program, but in which each main memory chip contains a processor. A program's data set is spread across these integrated processor/memory modules. All processors run the same program, broadcasting operands that they own to the other processors when needed, and performing any tasks that can be accomplished entirely on-chip without off-chip communication. This paper does not attempt a thorough, quantitative evaluation, but rather outlines a target system, pointing out some of the potential advantages over a conventional design.

In Section 2, we describe the DATASCALAR proposal, listing the four major advantages it has over traditional architectures, and describing how each advantage improves performance. In Section 3, we discuss implementation issues associated with these types of systems. In Section 4, we present a series of preliminary experiments and analyses that attempt to quantify the potential of the four major DATASCALAR advantages. Finally, in Section 5, we

1. 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.

list other research efforts related to processor/memory integration, present future directions, and conclude.

2 DataScalar architectures and SPSD

Given that each region of main memory is tightly integrated with a processor, new opportunities arise for achieving high performance. We are proposing to exploit the fact that all words in main memory have an on-chip processor. Solutions that require explicit parallelism are unacceptable—our goal is that existing serial programs should run without recompilation, and certainly without being rewritten. If this goal is achieved, new programs and compilers may of course exploit further opportunities posed by this novel architecture.

Our solution is an execution model for uniprocessor programs (an extension of Flynn's classification [12]) that is analogous to the Single-Program, Multiple Data stream (**SPMD**) execution model identified by Darema-Rogers *et al*. in 1985 [9]. This execution model, which we call Single-Program, Single Data stream (**SPSD**), was inspired by the Massive Memory Machine work from the early 1980s [15]. In **SPSD**, each of one or more processors runs the same program, reading and writing *exactly the same data* (unlike **SPMD**, in which each processor writes to different addresses).

A DATASCALAR system implements **SPSD** by having one or more integrated processor/memory modules (henceforth called *MOPs*, for *Memory On Processor*) run the same program, each MOP assuming ownership of the physical address space that it contains. When a MOP issues a load to an operand that it owns, it broadcasts that operand to the other MOPs (since they are all running the same program, they too will eventually need that operand). When a MOP issues a load to an operand that a different MOP owns, the load stalls, if necessary, until the needed operand arrives from the network, broadcast by the owning MOP. This ownership/broadcast scheme was called *ESP* by the Massive Memory Machine work.

We assume that off-chip communication will be comparatively more expensive in future systems (alternatively, we assume that computation will become less and less of a limitation). Forcing every load to be broadcast would therefore be a major drawback. We propose replicating the frequently-accessed portions of the address space both dynamically and statically, to cut down on inter-chip communication. For static replication, we duplicate the most heavily-accessed pages on each MOP, accesses to which will complete locally on every MOP, therefore not requiring a broadcast. Memory on each MOP is thus divided into two classes: *replicated* and *communicated*. A load to a *replicated* datum never requires a broadcast since it completes on every MOP, and a load to a *communicated* datum always requires a broadcast, since it completes only on the MOP that owns that particular datum. For dynamic replication, we allow each node to cache data owned by other MOPs. A load to a communicated datum that is found in all processor caches is not broadcast. This introduces some interesting consistency issues that we will discuss later in this paper.

Figure 1 shows a system-level comparison between a DATASCALAR system and a future system that has some on-chip memory but still has a large off-chip main memory. Although the replicated and communicated portions of main memory are depicted as separate units, they are only logically distinct.[1] The caches are not shown in this figure. Figure 2 shows how loads and stores to replicated versus communicated memory differ; both CPUs issue a load and store to replicated memory (L1 and S1), which complete on both MOPs. Both CPUs also issue loads to L2 and S2, which are located in the communicated memory of MOP-1 only. MOP-1 broadcasts L2, which MOP-2 receives and consumes. S2 completes at MOP-1, but is dropped at MOP-2.

The rest of this section describes the four categories of benefits that the DATASCALAR architecture provides.

2.1 ESP gains

The Massive Memory Machine (MMM) defined *ESP*, the notion of running the same program across multiple computational engines, broadcasting accessed local data to all non-local processors. However, the MMM proposed conventional, non-pipelined uniprocessors connected by a single global bus, and was therefore unlikely to provide better cost-performance than competing solutions. Furthermore, the MMM was a fully synchronous architecture, in which all processors proceeded in lock-step, with one processor running slightly ahead of the others (the *lead* processor). In Figure 3a we illustrate the high-level design of the MMM. In Figure 3b we show an example of the MMM's operation, in which processor 3 owns the first four operands, so is the lead processor for the first four accesses. Processor 2 owns operands five through seven, so upon the fifth access, a *lead change* occurs and processor 2 becomes the lead processor. Finally, another lead change occurs on the access to the eighth operand, and processor 3 again becomes the lead processor.

DATASCALAR systems enjoy the same benefits from ESP as did the MMM proposal. Major benefits are (1) reduced remote access latency, (2) elimination of request traffic, and (3) elimination of write traffic. Because each MOP runs the same program, a communicated operand can be sent to the other MOPs the instant its address is resolved and it is fetched from the on-chip memory. The request part of the access involves only an on-chip lookup. The operand is sent directly to the other MOPs, eliminating half of the communication delay by requiring only one-way communication. This "response-only" model also reduces traffic (increasing effective off-chip bandwidth) because off-chip requests are unneeded. Finally, all interchip write traffic is eliminated under ESP. Stores (or write-backs of dirty cache lines) complete locally on every MOP if they fall in a replicated page. Stores or write-backs to a communicated page occur only on the owning MOP, which preserves consistency since that MOP holds the only copy in main memory. Note that there are no consistency issues, because every MOP runs the same program.

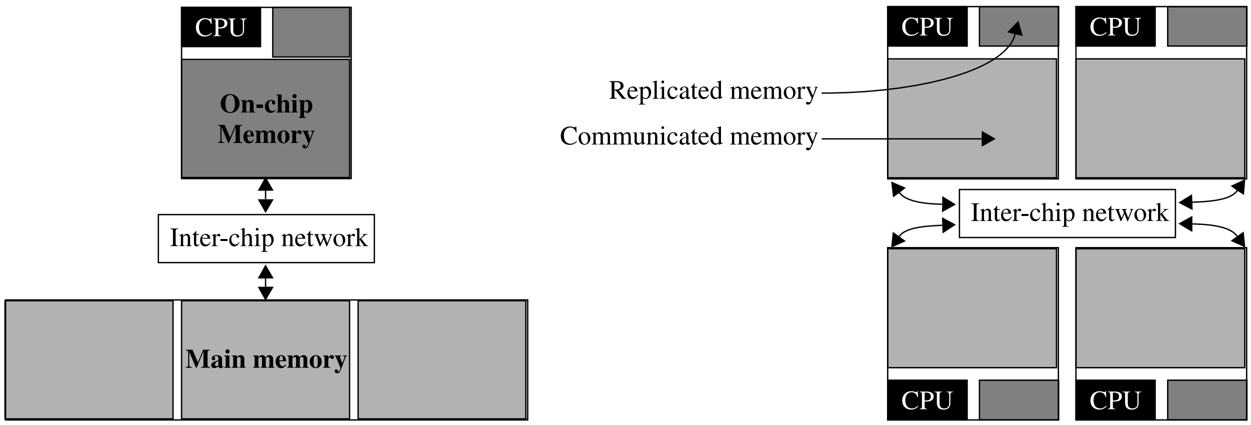
Since both the MMM and DATASCALAR systems implement ESP, they both enjoy these benefits of off-chip latency and traffic reduction. The next two subsections describe advantages that are unique to the DATASCALAR model.

2.2 Memory prefetching

Consider an access to a datum obtained through a pointer. In conventional systems, a request must be sent off-chip to memory, the pointer is returned, the processor computes the address of the datum, sends a request to memory, and the operand is returned. This sequence requires a total of four chip-to-chip crossings. An ESP-based system would incur two chip crossings at most: the owner of the pointer broadcasts the address, all nodes compute the address of the datum, and then the owner of the datum broadcasts the datum.

An ESP-based system such as DATASCALAR can do even better, however, if both the pointer and datum reside on the same MOP—the owner can therefore read both without waiting for an off-chip access, pipelining the broadcast of both operands to the other MOPs. We call the phenomenon of multiple consecutive accesses falling on the same MOP *memory prefetching*. Since each

1. For the purposes of this study, we have assumed a partitioning at the page level, and thus this distinction would be made in the page table. Other schemes are possible.



(a) Integrated architecture (traditional) **(b)** DATASCALAR architecture

Figure 1. Traditional system versus DATASCALAR system

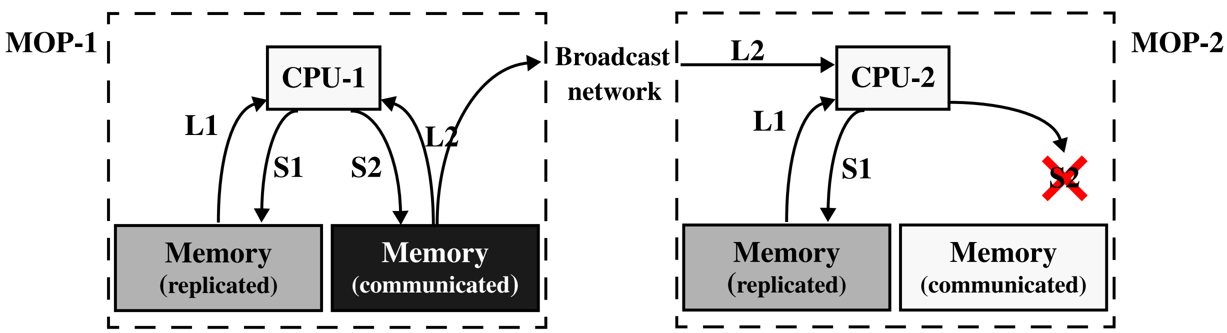


Figure 2. Replicated versus communicated main memory

memory chip has an on-chip processor, consecutive accesses falling on *any* memory chip will cause memory prefetching. Another way of visualizing memory prefetching is from the point of view of one MOP—from its perspective, it is the processor, and all other MOPs are simply memory—which can send it operands that it will need, before it has even computed their addresses.

Whenever an operand depends on another operand, and the two reside on different MOPS, an inter-chip communication is required, effectively halting any memory prefetching occurring down that dependence chain on any MOP. An example can be seen in Figure 3: if each w_{i+1} is dependent on w_i, there are only two inter-chip latencies on the critical path (after accessing w_4 and w_7). To increase the performance gains from memory prefetching, it is therefore desirable to maximize the number of consecutive references on single MOPs. We refer to the number of consecutive references to operands on a single MOP as a *streak*. A streak includes both replicated and communicated references.

With an in-order issue processor, a break in a streak will force the MOP to stall until another MOP broadcasts the needed operand. An out-of-order issue machine lends itself particularly well to this model, however, as multiple MOPs may prefetch down several dependence chains if the instruction window is sufficiently large. The ideal case is where all MOPs are memory prefetching down separate dependence chains that they contain locally.

The streak length does not always indicate the benefits of prefetching. For example, consider a program fragment that adds vector *A* to vector *B*, where all the elements of *A* reside on MOP *X* and the elements of *B* reside on MOP *Y*. If the code fetches alternately one element from *A*, then one element from *B*, the streak length will be very short. Processors that dynamically reorder instructions could easily race along in parallel, with processor *X* taking the lead fetching the elements of vector *A* while processor *Y* takes the lead in fetching the elements of vector *B*.

Memory prefetching does not require software support or recompilation—a DataScalar system may exploit spatial locality already inherent in reference streams. (Programs may benefit from recompilation or programmer tuning, of course, since explicit support could increase average streak length.) When streaks are greater than average, the DataScalar model benefits, since inter-chip latencies on the critical path are reduced. Memory prefetching does nothing to reduce bandwidth requirements, however, since the operands must still be broadcast to all MOPs. The next subsection describes how DataScalar systems may reduce inter-MOP traffic.

2.3 Result communication

DATASCALAR systems benefit from both ESP gains and memory prefetching without software support, but we believe that a significant potential for additional improved performance can be

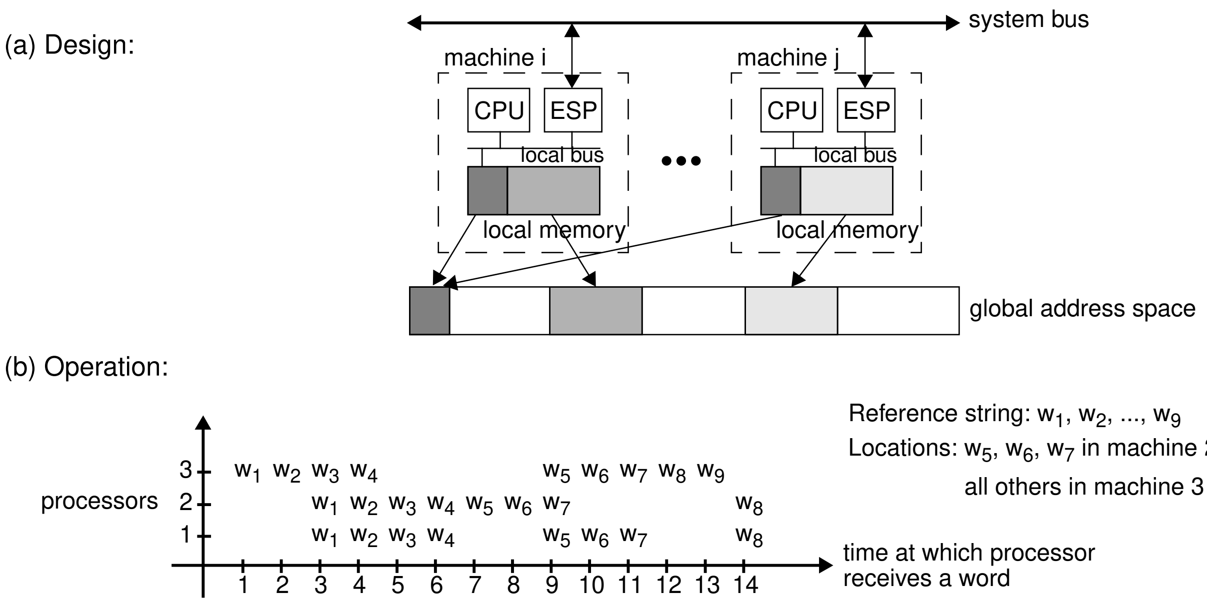


Figure 3. Design and operation of the ESP Massive Memory Machine (taken from [15])

achieved with compiler and/or programmer support. Another method of exploiting the fact that every memory chip contains a processor is *result communication*: when most or all of the operands for a computation are found locally, on one MOP, they are not broadcast: the result is computed locally only and broadcast to the other MOPs. If the result is to be written into the local communicated store (and is not likely to be needed again soon), the result store, and consequently the entire computation, can complete locally without incurring any off-chip traffic[1]. For example, if the program needed to sum an entire array, and store the result in the heap, having the entire array and heap element in communicated store on one MOP would enable the entire summation to be performed on-chip, including the final store. Subsequent accesses to that heap element would be correct, since only the owner would access the value from its communicated store and broadcast it. The software support would be needed to place the entire array on the same MOP, and to have the other MOPs bypass the code that is only to be run on the owning MOP.

The run-time system can guarantee that certain data are allocated on the same node, or the compiler can assume that no such guarantee exists. Optimizations are possible in either case; we provide specific examples in Section 4.3.

2.4 Exploiting coarse-grain parallelism

Although our DATASCALAR proposal focuses on running uniprocessor codes efficiently, a system with a processor on each memory chip is a *de facto* multiprocessor. Nothing precludes a DATASCALAR system from executing conventional parallel programs in a conventional way. In fact, a DATASCALAR system can switch between serial (ESP) modes and conventional parallel modes, exploiting easily-identifiable, coarse-grain parallelism when it exists, and running more or less serialized code efficiently when parallelism is hard to find. The result communication concept described in Section 2.3 is one manifestation of extracting parallelism from a traditional serial program. Many programs have some extractable parallelism, but at least some phases that achieve very poor speedup on conventional multiprocessors. These programs should benefit greatly from a DATASCALAR architecture. The issues associated with mixed serial/parallel mode execution are under investigation.

1. In fact, the other nodes would only compute the result, then throw it away!

3 DataScalar implementation issues

Because every operand must be present at every processor, the DATASCALAR scheme can only hope to succeed if most accesses can be found locally. At first glance, this would seem to be an impossible limitation, and these concerns cannot be easily dismissed. In particular, we see little hope that this technique can be extended to large numbers of MOPs. It is well known, however, that a large majority of memory references tend to access a small minority of the memory locations. For this reason, cache memories—particularly those specifically designed with this in mind—are often able to reduce remote accesses, sometimes dramatically [16, 4]. The challenge is to identify that part of the data and replicate it. Clearly, DATASCALAR must capture this locality in order to achieve the goal of minimal communicated data.

3.1 Dynamic Replication

Section 2 suggested that data can be replicated both statically and dynamically, and discussed the static replication of data. Dynamic replication can also be achieved, largely independent of the static methods employed, though the benefits of the two methods are unlikely to be fully additive. Dynamic replication can be achieved most easily by caching communicated data, turning it temporarily into replicated data. In Figure 4 we show an example of how such a system might split data into replicated and communicated classes. Some pages in main memory are marked as replicated, and some are marked as communicated. Some lines in the cache are from local communicated pages (marked as dynamically replicated), some are from local replicated pages, and some are caching communicated data owned by other MOPs (also marked as dynamically replicated).

The designation of data as replicated must be made with some care. In particular, it is necessary that communicated data be repli-

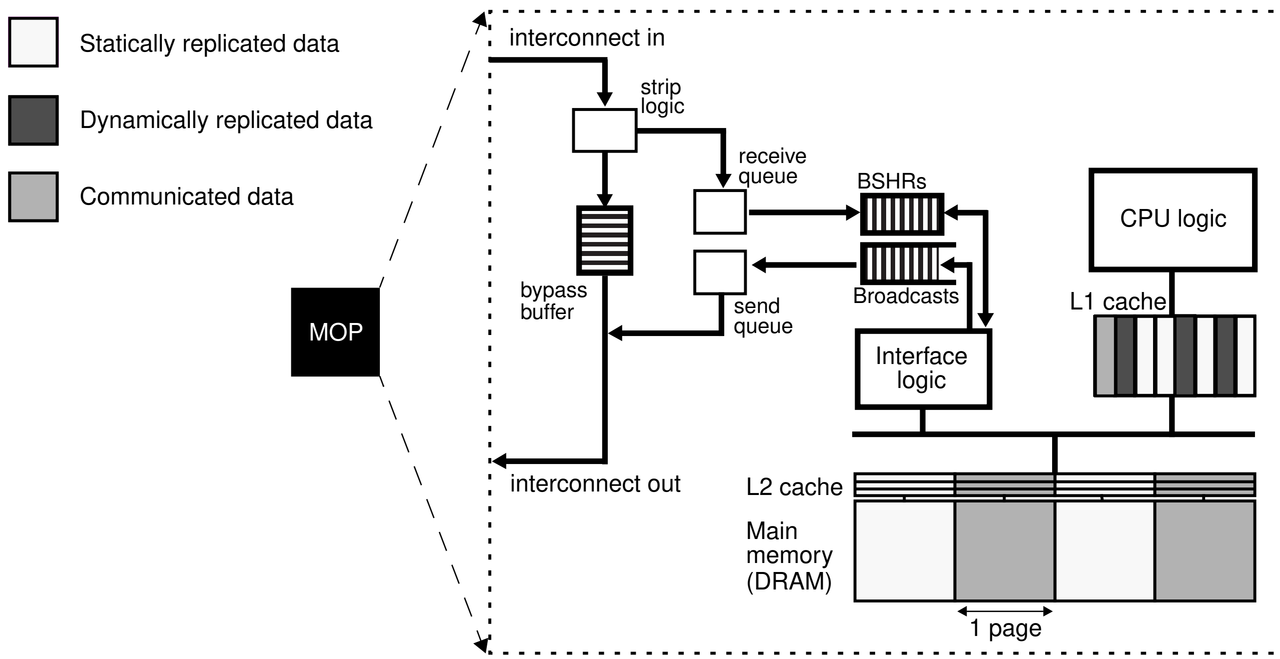


Figure 4. An example DATASCALAR machine

cated simultaneously at all nodes. This dynamic replication, however, creates some of the classic problems of shared-memory multiprocessors, requiring carefully defined cache- and memory-consistency models.

Because the system is running **SPSD**, however, many of the classic problems of memory consistency do not occur. There is no problem, for example, with different nodes trying to update the same memory location with different values. Care is required, however, to assure that all nodes receive every communicated datum exactly once. This can most easily be guaranteed by assuring that all caches maintain exactly the same set of communicated data. They may of course have independent replacement policies regarding the replicated data—all nodes have exactly the same replicated data and this is strictly a local decision—but they must all agree at all times on what items are replicated and what must be communicated.

The key notion is that all caches must make the same choices as to which communicated data to evict upon a replacement. Otherwise, the owner of a communicated datum might find the datum in its cache, and consequently refrain from broadcasting it. A different MOP that had evicted the line containing that datum would then never receive the needed datum. Out-of-order execution could lead to memory operations occurring in a different order, with a resulting difference in replacement decisions. For an access-based replacement policy (such as least-recently-used), one solution is to force accesses to the same cache *set* to occur in program order. For a fill-based replacement policy (such as first-in-first-out), the corresponding solution is to force *replacements* to the same set to occur in program order. Some first-level caches use a *random* replacement policy for speed. So long as inclusion is maintained, these issues can be ignored for all but the largest level of cache, which will presumably use a replacement policy other than random.

3.2 Inter-chip Communication

Because of the symmetric nature of the DATASCALAR model, all communicated values must be broadcast to all nodes. In general, broadcast operations are expensive, and clearly not scalable. In special circumstances, however, such as on a ring or bus, they may be accomplished with only minor additional cost, though reliable delivery and error recovery are inevitably more complicated for broadcast operations.

Ring operations, such as those defined in the IEEE standard Scalable Coherent Interface (SCI) [17, 28] seem particularly well-suited for this kind of operation. On a ring, all operations are observed by all nodes on a ring if the sender is responsible for removing its own message. We envision a ring interconnect because of the high-performance capability [26], but broadcast on a ring is complicated by the fact that operands originating at different MOPs are received at other nodes in different orders. A simple tag can sort out data to different addresses, but the issue becomes complicated when the same datum must be broadcast twice. Complications also arise whenever certain data items must be rebroadcast, or cancelled (due to full network queues or speculative broadcasts, respectively). The straightforward solution is to avoid broadcasting the same address a second time until all MOPs have accepted the first broadcast of the address. We are exploring other solutions as well, which use more sophisticated tagging of broadcasts.

In Figure 4 we show how the on-chip interconnect might interact with the off-chip network. The MOP contains a queue for broadcasts, which are obtained by interface logic snooping on the memory bus. The interface logic also inhibits the memory from servicing level-one cache misses and writebacks to non-local data. When a read to a non-local line occurs, an entry is allocated in the *Broadcast Status Holding Register* (BSHR), a structure similar to the MSHR proposed for lockup-free caches [20]. The major difference between the MSHR and the BSHR is that entries may be allocated in the BSHR by broadcasts arriving from the network, *before* the processor issues a request for that datum. When a network broadcast/processor request pair is matched, the BSHR forwards the data to the processor.

3.3 Speculative execution

Fine-grain speculative execution is now appearing in most state-of-the-art processors, and a successful DATASCALAR architecture must be compatible with out-of-order execution. Indeed, much of the promise we see in DATASCALAR is the opportunity for out-of-order execution, permitting several MOPs to race ahead simultaneously on different instruction sequences. However, speculation must be tightly controlled: the broadcast of data may well be a critical limitation of this model, and broadcast of data that goes unused will hinder performance in a bandwidth-limited system. We note that broadcasting data and then squashing it opens up many difficult problems for maintaining identical cache contents across nodes. While benefits may be seen from judicious broadcast of speculative data, the problems introduced seem too severe for the apparent payoff. We expect, therefore, that speculation will be restricted to local nodes' fetching their own data, whether replicated or communicated, and queuing up the communicated data for broadcast as soon as a speculative execution becomes non-speculative. We are exploring more aggressive speculative communication policies as well, however.

3.4 Operating Systems Issues

To the extent that an executing program is non-deterministic, operating system code can be executed in the same manner as user code. Synchronous exceptions, such as for an unaligned address, would be observed at slightly different times on different MOPs, but would cause no special problems. However, asynchronous events could potentially cause difficulty if they are not observed at the same point by all processors. Consider the case in which a write causes a page fault. Since only one MOP actually performs a write to communicated data—the other MOPs all simply discard their result—only the owning MOP would observe the page fault. If the other MOPs did not recognize the page fault, they might proceed beyond the fault point indefinitely. While it is interesting to consider such a variation on the idea of an imprecise exception, the problem can be avoided by making sure that all MOPs have the same page table entries, and actually check for exceptions on every memory operation. Thus each MOP would observe this page fault. External interrupts, likewise, must be injected into the system with care to assure that all processors observe them at the same point in their execution.

3.5 Cost

Modern computer systems, even high-performance systems, are becoming increasingly cost-sensitive. Thus the success of the DATASCALAR approach may depend largely on its cost. Because of the economies of scale, it is impossible to predict actual costs of future systems without knowledge of volumes, which are often largely unrelated to the effectiveness of the architecture or the quality of the design. We can only offer the relative costs of a DATASCALAR system as compared to a conventional processor.

It is our projection that with continuing advances, processors will become ubiquitous, appearing, even on DRAM chips to make more effective use of pin bandwidth. With each succeeding DRAM generation, the cost of an on-chip processor will become smaller, and in only a few generations, a moderately high-performance processor will be feasible at only a small incremental cost. An increasing portion of modern processors is devoted to memory, composing an increasingly sophisticated hierarchy of registers and levels of cache. We envision that this trend will continue, ultimately including DRAM as well, until latencies for off-chip accesses become so long (relative to instruction issue times) that these accesses must be treated more like page faults than cache misses. Such trends argue strongly for processor/memory integration.

Conventional systems today typically consist of a single processor and a collection of memory chips, with costs often split more or less equally between the two. A DATASCALAR system would consist of a collection of identical chips each of which costs more than a conventional DRAM chip, but less than a processor chip. Because such chips would have more memory than conventional processor chips, they should be able to achieve the same performance with less off-chip bandwidth, and therefore, fewer pins. Much more work is needed to characterize the communication and computation requirements of a DATASCALAR system before definitive cost/performance arguments can be made, but there is adequate reason to hope that such a system can be cost-competitive with more conventional systems of the future.

4 Measuring DATASCALAR system benefits

In this section, we discuss the potential of the previously enumerated, expected advantages of a DATASCALAR system. Since a full performance comparison between possible future systems is beyond the scope of this paper, we restrict our discussion to a high-level comparison of two system models: a centralized processor with an on-chip cache and off-chip main memory banks, versus a DATASCALAR system with small processors and caches on the memory system chips themselves.

4.1 ESP gain

The ESP broadcast model eliminates the need for off-chip request traffic and write traffic. In Section 3.2, we mentioned the additional cost of requiring broadcasts, which is dependent on the interconnect used in future systems. In this subsection we restrict our focus to traffic reduction.

Using execution-driven simulation, we measured the amount by which off-chip traffic, both in terms of bytes and transactions, would be reduced by using a DATASCALAR architecture instead of a traditional system design. We simulated fourteen of the SPEC95 benchmarks [31] using the SimpleScalar tool set [3], which simulates processors assuming a MIPS-like instruction set. The input sizes used were the `test` input sets for all benchmarks. We reduced the number of iterations for some of the benchmarks, after determining that the reduction did not qualitatively change the results. We assumed a 64-Kbyte, two-way set associative unified, write-allocate, write-back on-chip cache. 64-Kbyte is consistent with on-chip processor cache sizes at the time the SPEC95 benchmarks were written.

Table 1 shows the reduction in inter-chip traffic resultant from ESP, expressed in both bytes and transactions (we count a request/response pair as two transactions). The first column for each measure (bytes and transactions) shows the inter-chip traffic for a traditional system, as shown in Figure 1a[1]. The second column for each measure shows how much inter-chip traffic remains in a DATASCALAR system (Figure 1b), assuming no statically replicated data (which would correspond to on-chip memory in the traditional system). The third column for each measure shows how much of the original inter-chip traffic the DATASCALAR system still produces. What we see is that the ESP model eliminates roughly 15% to 50% of the inter-MOP bytes transmitted, and from 52% to 75% of the

1. Unlike Figure 1, however, we do not assume any on-chip memory other than the cache.

	Traffic(Mbytes)			Transactions (millions)		
Benchmark	**Total**	**DataScalar**	**Remaining**	**Total**	**DataScalar**	**Remaining**
tomcatv	1363	1150	84.4%	79.3	37.7	47.5%
swim	474	288	60.7%	27.6	9.5	34.2%
hydro2d	1182	794	67.1%	68.8	26.0	37.8%
mgrid	2371	1641	69.2%	138.0	53.7	39.0%
applu	588	363	61.7%	34.3	11.9	34.8%
m88ksim	41	35	85.7%	2.4	1.2	48.2%
turb3d	1539	920	59.8%	89.6	30.1	33.6%
gcc	1401	1129	80.5%	81.6	37.0	45.3%
compress	16	7	46.2%	1.0	0.3	26.2%
li	105	64	61.0%	6.1	2.1	34.3%
perl	1175	793	67.5%	68.4	26.0	38.0%
fpppp	15577	12992	83.4%	906.6	425.3	46.9%
wave5	1733	927	53.5%	100.9	30.4	30.1%
vortex	8971	7096	79.1%	523.6	232.5	44.4%

Table 1: ESP traffic reduction

individual transactions (because no requests are sent, the transaction reduction will always be 50% or greater). These results indicate that—for systems that spend much of their time stalled due to limited memory bandwidth—implementing ESP may improve performance or reduce system cost. Note that while these results are independent of the number of MOPs, and focus on traffic reduction, they do not address the performance penalty associated with requiring broadcasts to multiple nodes. We address this issue in Section 3.2.

The ESP model may argue for a cache write policy other than write-allocate, which seems to be an ill match for this model. Allocating a cache line on a store that will neither be read again soon, nor have neighbors that will be soon read, will incur a needless inter-chip broadcast. A better solution would be to have such stores bypass their caches, completing only at their owner (if their address resided in communicated memory; stores to replicated memory complete everywhere). Another alternative is to implement a write-validate policy [18], in which allocations are performed only when a load is issued to a line that is either not in the cache or does not contain the needed word (*i.e.*, its valid bit is not set).

4.2 Memory prefetching

In Table 2 we show experimental results that measure *streaks*—the number of consecutive references satisfied locally on one MOP—for a four-MOP system. These simulations also used the SimpleScalar tools and assumed an identical cache configuration to that presented in Section 4.1. For each benchmark, we replicated 32 4-Kbyte pages on each node. We selected the pages to replicate by running the benchmark, saving the number of accesses to each page, sorting the pages by number of accesses, and choosing the 32 most heavily-accessed pages. We distributed the communicated pages among the nodes round-robin in blocks with sizes ranging from 4 to 32 pages. The sizes of the distribution chunks are shown for each benchmark in the first column of Table 2. For each benchmark, we tried to maximize the distribution block size (to improve streak length) while still keeping it smaller than 1/4 of both the text and the largest data (globals, heap, stack) segments. This action prevented either segment from being completely contained on one MOP, making the streak length equal to the number of references.)

The next four columns in Table 2 show the distribution of replicated pages among the four segments. By comparing these with the sizes of each segment (shown in Table 3), the percentage of that segment that is replicated can be calculated. This quantity is relevant because greater replication within a segment tends to increase streak length.

The right-most four columns show the average streak lengths of four different types of streaks for each benchmark. The first calculates streaks using all references to memory (*e.g.*, all cache misses). The second and third columns compute streak length using only instruction and data references to memory, respectively. Finally, the right-most column shows the average number of contiguous accesses to replicated pages in main memory. Very high numbers of references to replicated pages will extend average streak lengths (a streak ends when a reference accesses a communicated page on a different MOP than the previous reference to a communicated page. If communicated references occur rarely, streaks will tend to be very long).

The average streak lengths in Table 2 tend to be very high for instructions—over 20 in every case. Part of this large length is due to the replication of text pages, which is significant for most programs (li, tomcatv, m88ksim, turb3d, and fpppp have average code streaks in the hundreds or thousands, and each has from 1/3 to 1/2 of the code replicated across all MOPs). Part of the explanation for the large streaks, however, is the high spatial locality generally found in code reference streams.

Data reference streak lengths tend to be lower than the instruction streak lengths. They are low (less than 3) for some of the floating point codes (swim, applu, turb3d, mgrid, and hydro2d). Although floating-point codes tend to have high spatial locality, streaks are cut by interleaved accesses to arrays residing on different mops (*e.g.*, `c[i] = a[i] + b[i]`). Also, some of the spatial locality is filtered out by the cache. The three other floating-point codes have higher average data streak lengths, however, ranging from about 6 to 33. The integer codes tend to have higher data streak lengths than do the floating-point codes. The data streak length for li is high because most of its data set is replicated. The others, however, do not have large fractions of their data set replicated, and they have average data streak lengths from about 3 to over 130.

Benchmark	Dist. size (Kb)	Replicated pages (128Kb)				Average streak length			
		text	global	heap	stack	total	text	data	repl.
tomcatv	32	22	6	2	2	42.3	31486.7	6.7	21.7
swim	32	7	24	0	1	2.1	60.2	2.1	1.0
hydro2d	32	25	5	0	2	1.7	176.9	1.6	1.1
mgrid	32	4	27	0	1	1.5	31.4	1.5	1.0
applu	32	23	8	0	1	2.6	43.3	2.6	1.0
m88ksim	64	16	10	5	1	157.3	859.2	69.1	16.2
turb3d	64	19	12	0	1	1.7	1541.6	1.6	1.1
gcc	256	25	1	0	6	7.4	23.9	4.5	1.2
compress	16	6	25	0	1	103.5	41.7	134.7	1.3
li	16	17	2	12	1	841.2	777.2	2027.1	208.4
perl	128	26	2	3	1	7.6	34.5	4.1	2.1
fpppp	64	27	4	0	1	165.6	755.9	33.7	3.7
wave5	64	17	14	0	1	6.4	171.6	5.9	1.7
vortex	128	27	2	1	2	5.5	21.0	2.9	1.9

Table 2: Streak measurements for a four-MOP system

Each row shows the experimental parameters for each benchmark, followed by the results. The first column contains the granularity at which communicated data are distributed round-robin around the MOPs. The second through fifth columns show the number of pages from each segment that were replicated for each benchmark. The right-most four columns show the average streak lengths for all reads, all reads to code and data, and reads to replicated memory, respectively.

Benchmark	text	global	heap	stack	total
tomcatv	164	28	37	14418	14647
swim	169	14421	28	10	14628
hydro2d	216	8653	44	14	8927
mgrid	174	7492	25	10	7701
applu	249	32322	27	27	32625
m88ksim	283	128	481	9	901
turb3d	246	25386	39	11	25682
gcc	2129	259	1694	309	4391
compress	103	43089	24	7	43223
li	178	21	88	9	296
perl	529	76	25613	8	26226
fpppp	341	475	26	21	863
wave5	389	41852	37	11	42289
vortex	970	127	25870	12	26979

Table 3: Data set sizes (in Kbytes)

Each row shows the breakdown of data set size for a benchmark, in kilobytes. The data set is broken down into the code, global data, heap, and stack segments. The right-most column displays the sum of the four components.

These results show that even with caches filtering out spatial locality, many programs will be able to take advantage of memory prefetching. DATASCALAR MOPs can run ahead of the others, finding multiple needed operands and instructions locally, and sending them to the other MOPs early—sometimes before the other MOPs have resolved the needed addresses.

4.3 Result communication

Although memory prefetching and the ESP gains are compatible with existing software, they do not aggressively exploit the notion of having a processor coupled with every memory chip. By moving the processing to the operands, instead of vice-versa (the traditional approach), we can exploit the integrated processor/memory model. Unfortunately, this approach cannot be done transparently (*i.e.*, without explicit software support) in a DATASCALAR system. When all operands needed for a computation are located on a single MOP, only the result need be broadcast (or stored). Optimizations are also possible when a majority of the operands needed for a computation reside on one MOP, but not all. It is here that the **`SPSD`** model becomes genuinely different from **`SISD`**: different MOPs may be following different paths of execution and issuing different instructions, but are still running the same program and working on the same data stream.

To allow different behavior on different nodes, the processors need an instruction that generates a value (or a condition code), which signifies whether a given address resides locally. This value could be used either as a branch condition or to support predicated execution, allowing different MOPs to engage in different behavior based on whether or not a datum is found on-chip.

The compiler, perhaps with programmer support, may compile for three different cases:

1. Operands for a computation are spread across multiple MOPs, with non-disambiguable dependences among them.
2. The operands are on multiple MOPs, but the operands on one of them are not needed to resolve the others.
3. All the operands can be guaranteed to reside on one MOP.

In case 1, the compiler does nothing, letting the system revert to transparent ESP execution. In case 2, the operands on all but one MOP may be broadcast, the one MOP performs the computation, and broadcasts the result to the other MOPs. This action reduces both traffic (the operands on the computing MOP are not sent) and computation (on the other MOPs). If a value on the computing MOP is needed by another MOP to resolve an address, this scheme fails, so the compiler must guarantee that this situation does not occur.

Disambiguating such dependencies at compile time is hard at best and impossible at worst. A simpler solution may be to guarantee that all of the operands needed for a computation reside on the same MOP. We can extend the virtual memory system to use certain bits of an address and a hash function to determine on which MOP it should place a given communicated page. We can then extend the run-time storage allocator to accept an address and return a pointer to allocated memory which resides on the same MOP as the given address. Coupled with loader support, related data structures may then be guaranteed to exist on the same MOP, or at least in the same MOP's virtual address sphere.

For brevity, we discuss only two examples below of how these techniques may be used, but many others exist.

Summing an array: in case 1 above, all MOPs containing elements of an array broadcast the elements that they own. Each MOP sums the entire array. An example of case 2 occurs if the MOP that owns the first array element did not broadcast the other elements it owns; instead, any MOP that did not own the first element would broadcast its elements but not perform the summation. The first-element MOP computes the sum, and broadcasts the result to all MOPs. In case 3, the array would be either linked to or dynamically allocated on the same MOP. The other MOPs would branch around the summation and receive the broadcasted sum. Other techniques from parallel processing, such as partial-sum reductions, could also be successful.

Manipulating a chained hash table: in case 1, lookups, insertions, and deletions all require inter-MOP communication. In case 2, when performing a lookup, traffic can be reduced by following an individual chain as long as the chain is local. After following its local section of a chain, that MOP broadcasts only the pointer that points to the subsequent element (which it does not own), so that the next MOP may begin to follow the chain. A benefit is only realized here when multiple consecutive elements in the chain reside on the same MOP. In case 3 (shown in Figure 5), chain elements are allocated in a heap page that resides on the same MOP as the head of the chain (using the run-time storage allocator modification described above). With this guarantee, the compiler can generate code that is bypassed by MOPs that do not contain the chain in their communicated store. When performing a lookup, the owner can race down the chain without waiting for an off-chip access, and broadcast the result of the lookup to the other MOPs. For insertions and deletions, no off-chip traffic occurs *at all*, since the result is written into the local communicated store of the owner. Note that while a conventional system may find part of the hash table on the processor, these optimizations always hold true for the *entire table*, since there is a participating processor on every memory chip.

Many other examples, both new ones and those drawn from the realm of shared-memory parallel processing, exist and can be exploited, given the appropriate level of programmer or compiler support. The extent to which compilers can identify opportunities for exploiting result communication *without* programmer support is unclear at present, however.

5 Conclusion

In this paper we have presented a system-level organization and execution model for future systems that have processors and memory coupled tightly together—a DATASCALAR architecture that runs the Single-Program, Single Data stream execution model. This proposal targets near-seamless expansion of highly-integrated systems, and is intended to benefit future systems that are limited by off-chip communication; *e.g.*, those that have a large disparity between the cost of an on-chip memory access versus that of an off-chip access. We break the potential benefits of this model down into four major categories, and discuss them along with the disadvantages of this architecture. We then discuss some of the issues associated with implementing this type of system. Finally, we provide measurements and discussion that indicate that there is indeed potential in the four benefit categories—ESP gains, memory prefetching, result communication, and hybrid parallel execution.

Many of our ideas were inspired by the Massive Memory Machine proposal, from which we obtained the concept of ESP [15]. Other research efforts are examining the running of uniprocessor programs much faster by using multiple program counters; the Multiscalar group at Wisconsin [13, 29] is one example. This is a complementary project, however, since we focus on the part of the system that is external to the processor (faster processors simply make our case stronger). Other projects are looking at processor/memory integration, such as the IRAM project at Berkeley [24], the PPRAM project at Kyushu University [23], and work at Sun Microsystems [25]. Also, Mitsubishi has developed a multimedia processor prototype integrated with on-chip DRAM [10].

This work is part of an ongoing research effort. We are in the process of building a detailed simulation infrastructure to evaluate DATASCALAR systems. We are also identifying compiler algorithms and language extensions that support the extraction of parallelism from **SPSD** (such as result communication). We are working to improve the DATASCALAR architecture itself—by supporting speculation more aggressively, improving static replication (perhaps on the level of objects or words), supporting coarse-grained dynamic replication (such as page promotion/demotion), and identifying techniques to maximize streak length.

The DATASCALAR architecture was originally conceived to permit system memory expansion in future systems that had integrated processors and memory. The goal was to be able to run uniprocessor programs efficiently and seamlessly, even given the presence of multiple processors on the memory chips. It is possible that the major benefit of DATASCALAR will be the ability to exploit parallelism in codes that were not traditionally thought of as candidates for parallel processing. Efficient serial execution, a seamless fallback case, and the notion of "memory parallelism" may enable levels of performance much than either current uniprocessors or parallel processors achieve alone.

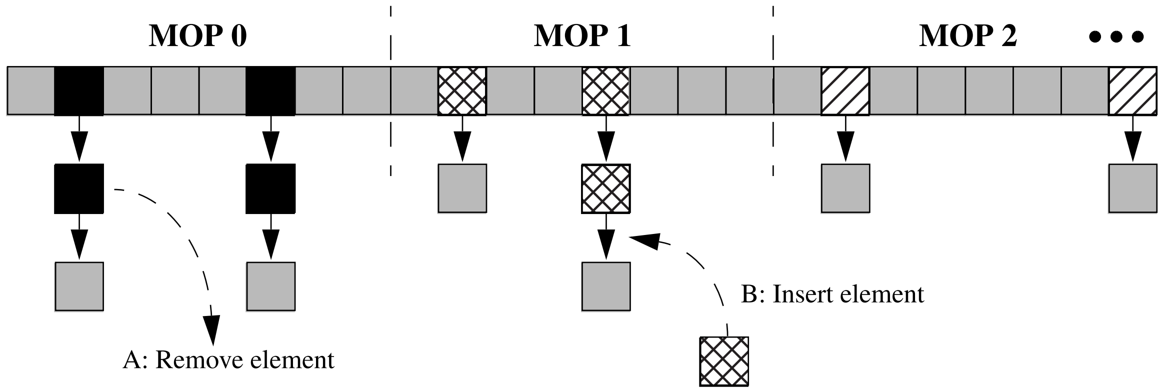


Figure 5. Operating on a fully-distributed chained hash table

Acknowledgments

The authors thank Alain Kägi, T.N. Vijaykumar, Scott Breach, and Babak Falsafi for their helpful discussions and intellectual contributions to this work, and Todd Austin, who developed the original SimpleScalar simulation tool set.

References

[1] Forest Baskett. Keynote address. *9th International Parallel Processing Symposium*, April 1995.

[2] Dileep P. Bhandarkar. *Alpha Implementations and Architecture: Complete Reference and Guide*. Digital Press, Newton, MA, 1996.

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

[4] Doug Burger, James R. Goodman, and Alain Kägi. Memory Bandwidth Limitations of Future Microprocessors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 79–90, May 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] F. Darema-Rogers, V. A. Norton, and G. F. Pfister. Using a Single-Program Multiple-Data Computation Model for Parallel Execution of Scientific Applications. IBM Research Report RC 11552, November 1985.

[10] Toru Shimizu et al. A Multimedia 32b RISC Microprocessor with 16Mb DRAM. In *Proceedings of the 1996 International Solid-State Circuits Conference*, pages 216–217. Mitsubishi Electric Co., February 1996.

[11] Keith I. Farkas and Norman P. Jouppi. Complexity/Performance Tradeoffs with Non-Blocking Loads. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 211–222, April 1994.

[12] Michael J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21:948–960, 1972.

[13] Manoj Franklin. *The Multiscalar Architecture*. Ph.D. thesis, University of Wisconsin, December 1993.

[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] 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.

[17] David V. James, Anthony T. Laundrie, Stein Gjessing, and Gurindar S. Sohi. Scalable Coherent Interface. *IEEE Computer*, 23(6):74–77, June 1990.

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

[19] 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.

[20] 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.

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

[22] 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.

[23] Kazuaki Murakami. PPRAM: A 21st Century's Microprocessor Architecture. Computer Architecture Seminar, UW-Madison, October 1995.

[24] David Patterson, Tom Anderson, and Kathy Yelick. The Case for IRAM. In *Proceedings of HOT Chips 8*, Stanford, California, August 1996.

[25] Ashley Saulsbury, Fong Pong, and Andreas Nowatzyk. Missing the Memory Wall: The Case for Processor/Memory Integration. In *Proceedings of the 23nd Annual International Symposium on Computer Architecture*, May 1996.

[26] Steven L. Scott, James R. Goodman, and Mary K. Vernon. Performance of the SCI Ring. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 403–414, May 1992.

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

[28] IEEE Computer Society. Scalable Coherent Interface (SCI). ANSI/ IEEE Std 1596-1992, August 1993.

[29] 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.

[30] 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.

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

[32] R. M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11(1):25–33, January 1967.