DataScalar Architectures

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

DataScalar architectures improve memory system performance by running computation redundantly across multiple processors, which are each tightly coupled with an associated memory. The program data set (and/or text) is distributed across these memories. In this execution model, each processor broadcasts operands it loads from its local memory to all other units. In this paper, we describe the benefits, costs, and problems associated with the DataScalar model. We also present simulation results of one possible implementation of a DataScalar system. In our simulated implementation, six unmodified SPEC95 binaries ran from 7% slower to 50% faster on two nodes, and from 9% to 100% faster on four nodes, than on a system with a comparable, more traditional memory system. Our intuition and results show that DataScalar architectures work best with codes for which traditional parallelization techniques fail. We conclude with a discussion of how DataScalar systems may accommodate traditional parallel processing, thus improving performance over a much wider range of applications than is currently possible with either model.

1 Introduction

Although microprocessor performance continues to grow at an exponential rate, not all microprocessor components improve similarly. Imbalances created by divergent rates of improvement are eliminated through microarchitectural innovations and altered assignations of resources. Specifically, the relative costs of communication are increasing relative to those of computation. This trend is resulting in successively larger caches with each processor generation, as well as more complex and expensive latency tolerance mechanisms.

As microprocessor clock improvements continue to outpace reductions in commodity DRAM access times and improvements in bus clocks, accesses to main memory grow more expensive. Techniques to reduce or tolerate this latency often do so by increasing the bandwidth requirements of the processor, which in turn increases the latency of memory accesses [4]. Processors that perform more parallel operations simultaneously may also increase queueing delays in the memory system.

This paper describes an architecture that exploits comparatively inexpensive computation to reduce communication overheads. In a DataScalar architecture, multiple autonomous processing units are each tightly coupled with a fraction of a program's physical memory. Each unit runs the same program, asynchronously executing the same instructions on the same data. When a processor issues a load to an operand that is located in its memory (i.e., the operand is *owned* by that processor), it broadcasts the result of the load to the other processors. When a processor issues a load to an operand not contained in its memory, the load request is buffered until the operand arrives, broadcast by some other processor. Stores are completed only by the owning processor, and dropped by the others.[1]

By performing redundant computation, a processor that has a datum locally can compute the address of that datum, access that datum, and send that datum to the other nodes quickly. Since all physical memory is local to at least one processor, a request for a remote operand never need be made. This execution model eliminates off-chip request and write traffic, reduces memory access latencies, and creates opportunities for new optimizations.

Current technological parameters do not make DataScalar systems a cost-effective alternative to today's implementations. For a DataScalar system to be more cost-effective than the alternatives, the following three conditions must hold: (1) Processing power must be cheap; the dominant cost of each node should be memory. (2) Remote memory accesses should be much slower than local memory accesses. (3) Broadcasts should not be prohibitively expensive.

We believe that technological trends are driving commodity systems in a direction such that the aforementioned conditions will all eventually hold. We list three possible candidates for DataScalar systems, ordered in increasing levels of integration, below.

- **Networks of workstations**: DataScalar execution could perform as an alternative to paging to remote physical memories [15] on a network of workstations, provided broadcasts were sufficiently inexpensive. Some network topologies, such as fat trees, support efficient broadcasts. Alternatively, some implementations of optical networks render broadcasts virtually free, enabling efficient DataScalar execution [2].
- **IRAM-based systems**: the concept of a single-chip computer, in which processor logic and main memory are merged on the same die has existed for decades. Processor/memory integration has received much attention recently [3, 20, 21, 17]; acknowledging this current enthusiasm, we refer to implementation of such systems as IRAM [20]. Remote memory accesses (to other IRAM chips) will certainly be more expensive than on-chip memory accesses. IRAM chips connected by a bus or point-to-point ring would exhibit the needed parameters for a cost-effective DataScalar implementation.
- **Chip multiprocessors (CMPs)**: Single commodity chips are projected to hold a billion logic transistors by 2010. Such copious silicon real estate will enable fabrication of single-chip multiprocessors containing substantial quantities of memory per processor—such a chip is in development even today [18]. Wiring delays

This work is supported in part by NSF Grant CCR-9509589, an unrestricted grant from the Intel Research Council, and equipment donations from Sun Microsystems.

1. Except when the data are cached, in which case the cache line is updated, and no write-through or write-back is required.

on these chips will be substantial [6]; accessing an operand from a bank to which the requester is tightly coupled is likely to be much faster than requesting an operand from a memory bank across the chip (in essence, the former access would be "local" and the latter "remote," even though both requests are serviced on the same chip).

In this paper, we evaluate a DataScalar implementation in the IRAM context, since we believe that it is a promising technological match for these ideas. In Section 2, we describe the ideas and background in more detail. In Section 3, we describe the benefits of the base model and present some supportive simulation results. In Section 4, we describe one implementation of a DataScalar system and present timing simulation results comparing it against a traditional alternative. In Section 5, we discuss the potential for the interaction of DataScalar and parallel processing. Finally, we discuss related work and conclude in Section 6.

2 DataScalar overview

DataScalar architectures are intended to mitigate the dual constraints of (1) processors coupled with limited memories that may not easily be expanded—such as a finite on-chip or on-module capacity in a highly integrated system—and (2) existing uniprocessor programs that are not easily parallelized. This architecture exploits the availability of multiple processors to minimize memory latency by using the fact that any memory location is local to *some* processor. Thus each read operand can be quickly fetched by some processor. Each memory update can be achieved by means of only one write by the processor to which the store address is local.

DataScalar is based on the Massive Memory Machine (MMM) work from the early 1980s. The MMM was a synchronous, SISD architecture that connected a number of minicomputers with a global broadcast bus [13]. Each computer contained a very large memory, which was a fraction of the total program memory (each operand was thus *owned* by only one processor, i.e., it resides in the physical memory of only one processor). All computers ran the same program in lock-step, and the owner of each operand broadcast it on the global bus when accessed. This broadcast model was called *ESP* in the MMM work. We depict an example of synchronous ESP in Figure 1. One processor (the *lead* processor) is slightly ahead of the others while it is broadcasting (initially processor 3 in Figure 1). When the program execution accesses an operand that the lead processor does not own, a *lead change* occurs. All processors stall until the new lead processor catches up and broadcasts its operand (e.g., processor 2 broadcasting w_5 at cycle 7 in Figure 1).

Some advantages to ESP are: (1) that no requests need be sent, thus reducing access latency and bus traffic, since all communication is one-way. (2) Writes (or write-backs) never appear on the global bus, further reducing bus traffic (since all processors are running the same program, they all generate the same store values, which need complete only on the owning processor). (3) Since the MMM was synchronous, and all processors generated the address for each successive operand, no tags needed to be sent with the data on the global bus.

DataScalar architectures combine ESP with out-of-order execution, the combination of which is an asynchronous version of ESP. Each processor may access owned operands simultaneously. This asynchrony permits each processor to run ahead on computation involving operands that it owns, generating the total stream of broadcasts more quickly. We call this capability *datathreading*. Unlike synchronous ESP, however, each broadcast must contain an address or tag, since broadcasts occur in an unknown total order.

Because each processor executes the instructions in a different order, it is possible for a processor to temporarily deviate from the ESP model and execute a private computation, broadcasting only the result—not the operands—to the other processors. We call this technique *result communication*, and discuss it in more detail in Section 5.1. We note that this execution model—which we call Single-Program, Single Data stream (SPSD)—is a serial analogue to the Single-Program, Multiple Data stream (SPMD) execution model proposed by Darema-Rogers et al. in 1985 [5] (which extended Flynn's classification [11]).

Requiring every load to be broadcast would generate much more total traffic than current systems with cache memories. Inter-chip traffic can be reduced dramatically by replicating the frequently accessed portions of the address space both statically and dynamically at various granularities (word, cache line, and/or page).

We replicate data statically by duplicating the most heavily accessed pages[1] in each processor's local memory. Accesses to a replicated page will complete locally at every processor, requiring no off-chip traffic. The address space is thus divided into two categories: *replicated* and *communicated*. Replicated pages are mapped in each processor's local memory, and the communicated part of the address space is distributed among the processors.

Figure 2 shows how loads and stores differ for replicated and communicated memory; both processors issue a load and store to replicated memory (L1 and S1), which complete locally. Both processors also issue loads to L2 and S2, which are located in the communicated memory of processor **A** only. Processor **A** broadcasts L2, which processor **B** receives and consumes. S2 completes at processor **A**, but is dropped at processor **B**.

Static, coarse-grained replication of pages cannot capture locality that is fine-grained or identifiable only at run-time. We must therefore allow caching at each node, effectively replicating data dynamically for the period that they are cached. Dynamic replication of data, however, introduces some new consistency issues that we will discuss in Section 4.1.

The DataScalar execution model is a memory system optimization, not a substitute for parallel processing. When coarse-grain parallelism exists and is obtainable, the system should be run as a parallel processor (since a majority of the needed hardware is already present). We discuss the issues concerning hybrid execution models for multiprocessors further in Section 5.2.

3 DataScalar benefits

In this section, we describe the benefits associated with the base DataScalar model (ESP and datathreading) in detail. We show how ESP reduces about a third of off-chip traffic (on average), and we show how datathreading offers the potential for reductions in memory latency. We also present simulation results that address each of these benefits.

3.1 ESP and traffic reduction

DataScalar systems enjoy nearly the same benefits from ESP as did the MMM proposal. ESP reduces traffic—thereby increasing effective bandwidth—by eliminating both request traffic and write traffic from the global interconnect. ESP, asynchronous or otherwise, does not further reduce the number of read operands that must be communicated off-chip over that of a conventional architecture.

ESP-based systems eliminate request traffic because ESP uses a *response-only* (or *data-pushing*) model. Since all processors run the same program, if one processor issues a load to an address, all the other processors will eventually issue that same load. The owner is therefore assured that when it broadcasts the load, all

1. We assume a static partitioning at the page level, and thus this distinction would is in the page table. Other schemes are possible.

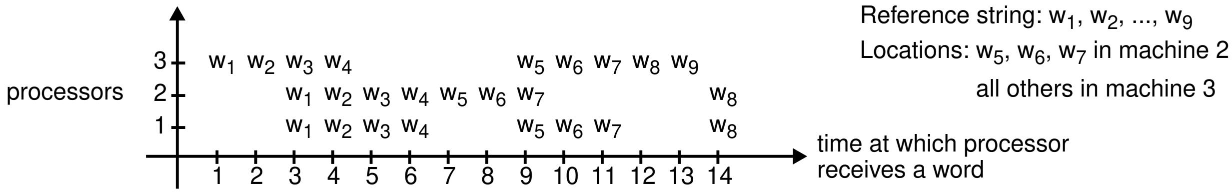


Figure 1. Operation of the ESP Massive Memory Machine (from [13])

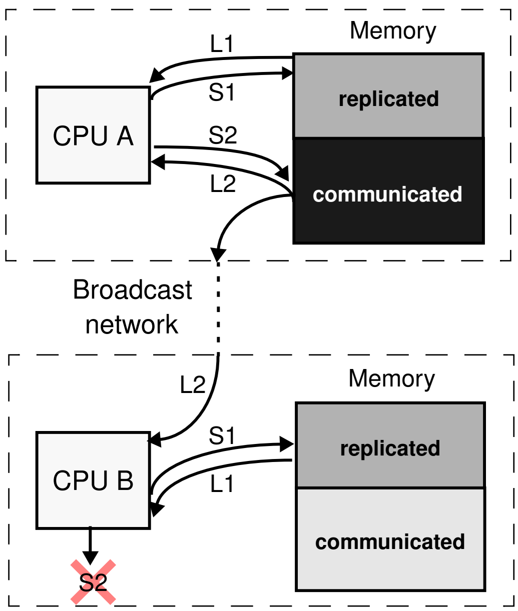


Figure 2. Replicated vs. communicated memory

other processors will consume it. Conversely, when a processor issues a load to a datum that it does not own, it can buffer the request on-chip, and the matching data will eventually arrive. Thus, requests need never be sent off-chip. Similarly, when a store is generated at all nodes, only the owner of that address need complete the store on-chip. Since every chip is generating the value locally, created store values never need be sent off-chip. All processors will complete the store if the address is a replicated location. If the address is cached at all nodes, the store will complete in the cache, and the eventual write-back (or write-through) operation will be dropped at nodes that do not own that address. Note that there are none of the traditional cache consistency issues, since every processor is running the same program.

In a synchronous implementation of ESP, tags need not be broadcast with data—every processor is generating the same instruction stream in the same order, so tags can be inferred from the order in which the broadcasts are received. DataScalar systems do not enjoy this benefit; the out-of-order issue processors will all issue multiple broadcasts in an unpredictable order. In addition, more than one processor generally will be attempting to broadcast at any given time. This lack of predictability means that data must be broadcast along with their addresses and/or some other identifying tags (multiple instances of the same address may require supplementary tag information, such as a sequence number).

We measured the extent to which ESP reduces off-chip communication using cache simulation. We used the SimpleScalar tool set [1], which is an execution-driven set of processor simulators that simulate a MIPS-like instruction set architecture. We simulated a 64-Kbyte, two-way set-associative, write-allocate, write-back, on-chip level-one data cache (this size is consistent with typical cache sizes at the time that SPEC95 was released). We measured the aggregate miss traffic from the cache, and calculated the fraction of traffic that remained once write-backs and requests were eliminated. In Table 1, we show this measured fraction for fourteen of the SPEC95 benchmarks. We show both total traffic eliminated, and the reduction in the total number of distinct messages (we count a request/response pair as two transactions). The table shows that, for this cache size, ESP eliminates roughly 15% to 50% of the off-chip traffic in bytes, and from 52% to 75% of the individual transactions (because no requests are sent, the transaction reduction will always be at least 50%).

These results indicate that—for systems in which memory bandwidth is at a premium—implementing ESP is likely to improve performance, or reduce the required system cost to achieve the same performance. These results focus solely on bus traffic reduction—they do not address the performance penalties associated with necessitating broadcasts on interconnects other than buses. We address that issue in Section 4.4.

3.2 Datathreading and latency reduction

ESP-based systems reduce memory latency by making all off-chip communications one-way only. These savings might be large if the remote communication time dominates the memory request latency, or small if the memory access latency and/or memory system queueing delays dominate the request latency.

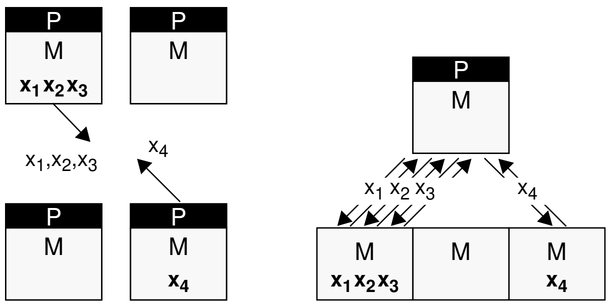
ESP-based systems offer the potential for further reductions in memory access latencies, however. Consider a stream of accesses to memory locations, each address of which is dependent on the value of the previous address (e.g., pointer chasing). When two or more dependent addresses reside in one processor's local memory, that processor may fetch those values without incurring any off-chip latencies. Those values may then be sent to the other processors by pipelining the broadcasts, incurring only one off-chip delay on the critical path. All processors thus complete the processing of those addresses faster than would a traditional system.

To illustrate this concept, we depict a simple example in Figure 3. Figure 3a shows a four-chip DataScalar system in which each IRAM chip contains a quarter of the program's physical memory. Figure 3b shows a more traditional organization, in which one IRAM chip holds a quarter of the program's memory and traditional DRAM chips hold the other three-quarters. In both systems, operands x_1, x_2, x_3 all reside on one chip, and operand x_4 resides on a different chip. The address of each x_{i+1} is dependent on x_i. One processor in the DataScalar system can access the first three without a single off-chip access, and then pipeline the broadcasts of those three operands to the other nodes (the broadcasts will be separated by the memory access time, of course). There will be a serialized off-chip access between x_3 and x_4 (analogous to a lead change in the MMM), and then x_4 will be broadcast. The system thus incurs two serialized off-chip delays. The traditional system, conversely, incurs two serialized off-chip accesses (one request, one response) for each operand, for a total of eight in this example The traditional system would incur zero off-chip delays if all the

Quantity eliminated:	tomcatv	swim	hydro2d	mgrid	applu	m88ksm	turb3d	gcc	compress	li	perl	fpppp	wave5	vortex
Traffic	.16	.39	.33	.31	.38	.14	.40	.19	.54	.39	.32	.17	.46	.21
Transactions	.52	.66	.62	.61	.65	.52	.66	.55	.74	.66	.62	.53	.70	.56

Table 1: Off-chip data traffic reduced by ESP

Shows reductions in off-chip data traffic due to ESP (removal of write and request traffic) for the SPEC95 benchmarks. Traffic is measured in two ways: fraction of bytes eliminated (top row) and fraction of transactions eliminated (second row).



(a) DataScalar system:
Pipelined broadcasts
Serialized off-chip accesses: 2

(b) Traditional system:
Req./response each operand
Serialized off-chip accesses: 8

Figure 3. Comparing off-chip access serializations

operands happened to reside in the on-chip quarter of the memory, as opposed to a minimum of one for a DataScalar system.

We call a series of accesses to consecutive local dependent operands a *datathread*. If the operands are not dependent, then a traditional system could simply pipeline multiple non-blocking accesses, obtaining them in two serialized off-chip crossings. When a dependence spans two nodes, we view that point as initiating a datathread migration from one node to the other, beginning the access stream of that thread at the new node. The overhead of migrating this conceptual thread is one serialized off-chip access. The cost of maintaining inexpensive datathread migrations is precisely that of maintaining SPSD execution— broadcasting loads and performing computation redundantly at all nodes.

Another conceptual view of asynchronous ESP execution is that from each processor's perspective, it is the main processor, and the others are simply intelligent prefetch engines residing in the main memory modules. From this perspective, the broadcasts the processor sends are merely the state the prefetch engines need to continue performing the accurate prefetching. Since this is a homogenous system, each processor will have this view of the others, of course.

The Massive Memory Machine was able to exploit only one datathread at any time; when a lead change occurred, a new datathread began at the new leader (in Figure 1, operands w_1-w_4, w_5-w_7, and w_8-w_9 would constitute three datathreads, assuming each operand is dependent on the previous one). DataScalar systems, because they implement asynchronous ESP with out-of-order issue at each node, may have multiple datathreads running concurrently. DataScalar systems do not require special support for datathreads, since they transparently exploit the locality already inherent in reference streams. However, programs would benefit from special support to increase datathread length or raise the number of datathreads executing concurrently.

In Table 2 we show experimental results that measure the mean number of loads falling consecutively on a single node. This is an approximation of datathread length, since we do not account for dependences. All results presented here assumed a four-processor system. These simulations also used the SimpleScalar tools and assumed a cache configuration identical to that presented in Section 3.1. 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 distributed blocks of data 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 datathread length) while still keeping it smaller than 1/4 of both the text and the largest data (globals, heap, stack) segment. This action prevented either segment from being completely contained at one processor, a situation which would make the datathread length equal to the number of references.)

The next four columns in Table 2 show the distribution of replicated pages among the four segments. Columns seven through nine show the mean (arithmetic) datathread lengths using three different definitions of datathreads. All three methods count consecutive references on a node, beginning the count upon the first reference to a communicated datum local to some node, ending (and restarting) the count upon the next reference to communicated data local to a different node. Column seven approximates datathread lengths using all references to memory (e.g., all cache misses). The second and third columns compute datathread length using only instruction and data references to memory, respectively.

The right-most column shows the average number of contiguous accesses to replicated pages in main memory. High numbers of references to replicated pages will extend average datathread lengths. If references to replicated data are frequent, the threads will tend to be long.

The average datathread lengths in Table 2 are generally high for instructions—over 20 in every case. These large numbers are partially due to the replication of a high percentage of the text pages, which is significant for most programs (`li`, `tomcatv`, `m88ksim`, `turb3d`, and `fpppp` have average code datathreads in the hundreds or thousands, and each has from 1/3 to 1/2 of the code replicated across all processors). Part of the explanation for the long datathreads, however, is the high spatial locality generally found in code reference streams.

Data reference thread lengths that we see tend to be shorter than the instruction thread 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, our approximation of datathreads is cut by interleaved accesses to arrays residing at different processors (*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 datathread lengths, however, ranging from about 6 to 33. The integer codes tend to have higher datathread lengths than do the floating-point codes. The datathread length for `li` is high because most of its data set is replicated. The others show average datathread lengths from about three to over 130.

These results show that many programs will be able to exploit datathreading. Ideally, each processor in a DataScalar system will run ahead of the others, finding multiple needed operands and

Benchmark	Dist. size (Kb)	Replicated pages (128Kb)				Datathread length approximation			
		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: Approximate datathread measurements for a four-processor 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 processors. 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 arithmetic mean of our datathread length approximations for all reads, all reads to code and data separately, and reads to replicated memory, respectively.

instructions locally, and sending them to the other processors early—sometimes even before the other processors have resolved those addresses.

4 Performance of a DataScalar implementation

In this section we present simulation results comparing one implementation of a DataScalar architecture with a more traditional architecture. We first discuss the specific solution we implemented to enable caching under asynchronous ESP. We then describe our simulated architecture, simulation environment and parameters, and present our results.

4.1 Cache correspondence

In Section 2 we described static replication of data, in which heavily used pages are copied at each processor running as a DataScalar machine. Static replication is limited in that it cannot use run-time information to reduce off-chip accesses—caches are universally used precisely because this run-time information is so crucial. Dynamic replication, therefore, is crucial to the competitiveness of DataScalar systems.

Dynamic replication in a DataScalar system is analogous to caching in a uniprocessor; processors take a broadcast operand or block of data, and decide to cache the data locally for a period of time (the difference is that multiple processors are all caching the same data instead of just one). However, replicating data dynamically is more complicated than simple caching. The goal of replication is to improve average memory access latency by reducing the number of broadcasts (which are analogous to cache misses in a uniprocessor). If the owner of a datum decides not to broadcast it upon a load, assuming it to be replicated, *every other node must still have that operand*, or deadlock will result. Conversely, if the owner broadcasts the operand and other nodes already have that operand locally, superfluous messages may fill up the queues on the remote nodes (depending on the broadcasting/receiving implementation). Certainly unnecessary broadcasts will waste bandwidth.

All nodes in a DataScalar system must therefore keep exactly the same set of dynamically replicated data, all choosing to stop replicating a datum at the same point in the access stream. Furthermore, these nodes should ideally make the decisions about what to keep replicated and what to throw out based on *local information only*—requiring continuous remote communication solely to reduce the number of broadcasts would make DataScalar systems non-competitive.

While many solutions are conceivable, in this paper we describe only the one that we have implemented. Our solution is to fold the decisions about what to replicate dynamically into the first-level caches—a block is considered to be dynamically replicated so long as it is in those caches.[1] If a level one cache miss occurs for communicated data, the owner must broadcast that line to the other nodes. This solution implies that no node may ever miss on a communicated line if another node hits on that line for the same load. We call this the *cache correspondence* problem; data must be kept *correspondent* in the primary caches to prevent deadlocks.

Keeping the caches correspondent is a non-trivial problem. Dynamically scheduled processors will send loads to the cache in different orders, and will also send different sets of instructions (when branch conditions take longer to resolve at some processors than others, allowing more mis-speculated instructions to issue). If two loads to different lines in the same cache set are issued in a different order at two processors, that set will replace different lines, and the caches will cease to be correspondent.

Our solution is to update the primary cache state only when a memory operation is *committed*, not when it is issued. To maintain correct program semantics, instructions must be committed in the same order at all processors, even though they may be issued in different orders. This solution also prevents mis-speculated instructions from affecting the cache contents.

We implement this solution with a structure called a *Commit Update Buffer* (CUB). We envision separate CUBs for instructions and data (ICUBs and DCUBs), but in this paper we only evaluate a DCUB. When a cache miss returns, rather than loading the data

1. It is possible to use lower levels of a multi-level cache hierarchy to perform dynamic replication. We chose to use only the level-one caches because our particular solution requires a tight coupling of the cache tags and the load/store queue in the processor.

into the cache, the line is placed into an entry of the DCUB, and a pointer to that entry is placed in the load/store queue at the entry of the load that generated the miss. Memory operations to the same line are serviced by the data in the DCUB (loads may still be serviced by stores farther ahead in the load/store queue). When a memory operation is committed, the cache tags are updated, and, if necessary, the line is loaded from the DCUB into the cache. A DCUB entry is deallocated when the last entry in the load/store queue that uses that line is committed. In addition to a pointer to the DCUB entry, each entry in the load/store queue is extended with state that represents whether the instruction missed in the primary cache at issue time.

This extra state is necessary because updating the cache at commit time only is sufficient to guarantee cache correspondence, but not to guarantee identical hit/miss behavior at all processors. Since instructions may issue at different times across processors, the same instruction will issue at different commit points in the instruction stream across the processors, causing some to hit and others to miss in their caches. By saving whether a hit or miss occurred at issue time, we can compare that event with the correct commit-time event, and take corrective action if there is a disparity.

We show a simple example in Figure 4. Two addresses, **X** and **Y**, conflict in the cache. Instructions commit from left to right. The second load to **X** ($\mathbf{X_2}$) misses when issued, but would have hit at commit time if the instructions were issued in program order (because $\mathbf{X_1}$ would have already generated the miss). This is an example of a *false miss*. Analogously, $\mathbf{Y_2}$ hits at issue time because $\mathbf{Y_1}$ had just been committed, but should have missed at commit time (e.g., at another processor, $\mathbf{Y_2}$ might issue after $\mathbf{X_1}$ is committed, causing a miss at issue time instead of a hit). We call this a *false hit*, and deal with it by generating a reparative miss when this situation is detected at commit time (a reparative miss consists of a reparative broadcast by the owner, or a squash to the local BSHR by a non-owner of that datum). We deal with false misses by recognizing that any sequence of accesses to the same line will generate only one miss ($\mathbf{X_1}$ and $\mathbf{X_2}$ in this example). If $\mathbf{X_1}$ issues after $\mathbf{X_2}$, we can "assign" the miss generated by $\mathbf{X_2}$ to $\mathbf{X_1}$, thus ensuring that all processors will generate only one miss for that line.

This "cache correspondence protocol" does not currently handle speculative accesses; if we were to permit incorrect speculations in our simulations, we would have to buffer speculative broadcasts at the network interface. We would then allow them to proceed only when they were determined to be correct, and squash them locally otherwise. We are in the process of extending this correspondence protocol to support speculative broadcasts.

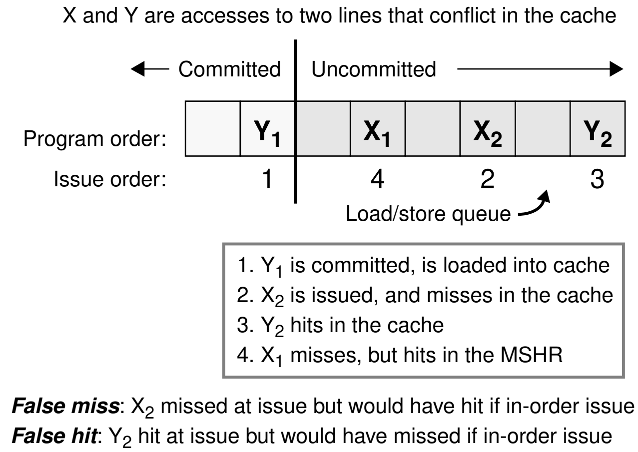


Figure 4. Cache correspondence example

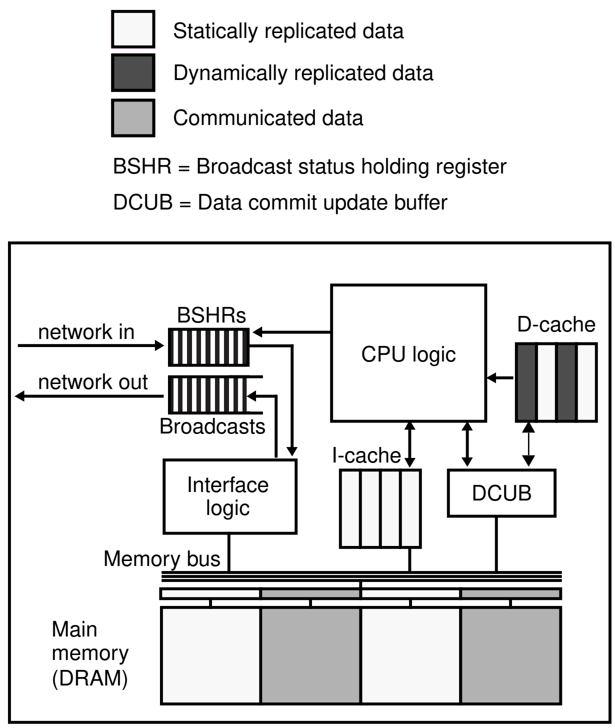


Figure 5. Simulated DataScalar chip datapath

4.2 Simulated implementation

We evaluated a DataScalar system consisting of multiple integrated processor/memory (IRAM) modules connected via a global bus. In Figure 5 we show a diagram of the high-level datapaths present in our simulated DataScalar implementation. We assume split primary instruction and data caches. We replicate the program text at each node, obviating the need for dynamically replicated instructions (and therefore a speculative correspondence protocol). We do support dynamic replication of data, so a DCUB, not the accesses themselves, updates the data cache tags and storage. We assume a fast on-chip main memory, which is insufficiently large to hold an entire program data set, but which is fast enough to eliminate the need for a level-two cache.

We use a simple queue to buffer broadcasts being placed on the global bus. The process of receiving broadcasts is more involved. We call the broadcast-receiving structures that we simulate *Broadcast Status Holding Registers*, or BSHRs. We implement the BSHRs as a circular queue. When a broadcast arrives from the network, the BSHR performs an associative search on that address. If a match occurs, the earliest entry matching that address in the queue is freed and the data are forwarded to the processor. If no match occurs, the BSHR allocates the next entry in the queue and buffers the data. In this case, when the processor issues the request for the data, it finds them waiting in the BSHR, and effectively sees an on-chip hit.

Level-one cache misses become broadcasts if the missing cache line is in communicated memory, and the processor is the owner of that cache line. The miss allocates a BSHR entry if, at a given processor, the miss is to a line that is both communicated and unowned by that processor. In Figure 5 we show a datapath from the processor to the BSHR queue; this path is used to squash BSHR entries allocated due to false misses.

Our simulation platform was a substantially modified version of the SimpleScalar tools [1]. To simulate DataScalar systems, we extended the SimpleScalar out-of-order processor simulator with multiple target contexts. The simulator switches contexts after executing each cycle (i.e., it simulates cycle n for all contexts before simulating cycle $n + 1$ for any context).

We also implemented address translation, which was not present in the original version. We assume a single-level page table, locked in the low region of physical memory. We maintain the replicated vs. communicated state of each page with a bit in the page table entry. Each page table entry also has one bit that determines ownership of a communicated page (only one processor will have the ownership bit set for a communicated page; the bit for that page is cleared in the page table entries of all other processors).

Processor parameters

Evaluating future systems, particularly those five or ten years away, is always difficult. Simulating the processors of tomorrow on the machines of today (using the benchmarks of yesterday) makes choosing parameters that give meaningful results difficult. We opted for an aggressive processor model, coupled with a memory hierarchy that has cycle times matching the generation of our hypothetical future processor, but which is sized according to the year that the benchmarks were released.

For all our experiments, we targeted a processor that might be built about five years hence. We assumed an 8-way issue, 1 GHz, out-of-order issue processor. Our processor used a Register Update Unit (RUU) [24] to keep track of instruction dependencies. We simulated an instruction window size of 256 instructions (RUU entries). Our simulated processor also contains a load/store queue, to prevent loads from bypassing stores to the same address. Loads are sent from this queue to the cache at issue time, while stores are sent to the cache at commit time. Loads can be serviced in a single cycle by stores to the same address that are ahead in the queue. For all simulations, we simulated a load/store queue that had half as many entries as did the simulated RUU.

Modern branch predictors are already quite accurate, however, and we have no way of knowing what prediction techniques will be prevalent in future processors, or the extent to which these processors will engage in aggressive speculation. We therefore assumed perfect branch prediction in our simulations. This assumption simplified our handling of the BSHRs (since our cache correspondence protocol does not currently support speculative broadcasts). Assuming perfect branch prediction will also increase the measured IPC, due to the absence of branch misprediction penalties (the IPC of future processors is likely to be even higher as they engage in speculation that is much more aggressive than branch prediction [25]).

Memory system parameters

On-chip memories are likely to be significantly faster than DRAMs are today. Using sub-banking, with hierarchical word- and bit-lines, will enable DRAM banks to have access latencies that are comparable with those of cache memories. Current high-density (1 Gb) DRAM prototypes, the processes of which are optimized for density and not speed, have access latencies in the low 30's of nanoseconds [7, 8]. On-chip DRAM banks implemented in hybrid memory/logic processes are likely to be significantly faster.

For our simulations, we assume a memory hierarchy on-chip that is just two levels. The first level is split instruction and data caches, 64KB each with single-cycle access. The caches are direct-mapped (for speed) and the data cache implements a write-back, write-noallocate policy. We believe that this write policy is superior to write-allocate in an ESP-based system (with a write-allocate protocol, a write miss requires sending an inter-processor message, only to overwrite the received data). Both caches are fully non-blocking and can support an arbitrarily high number of outstanding requests. The second level of the hierarchy is composed of high-capacity, on-chip memory banks that can be accessed in 8 ns. They are connected with a 256 bit bus that is clocked at the processor frequency. We assume that our off-chip bus is 128 bits wide and is clocked at 200 MHz. Commodity parts that expect to do most of their computing and memory accesses on-chip are not likely to have support for extremely aggressive off-chip connections.

We assume BSHRs with 3-ns access latencies and 128 entries. We assume a broadcast queue for the DataScalar simulations, which incurs a two-cycle access penalty before broadcasting data onto the global interconnect (the traditional architecture, similarly, buffers off-chip requests at a network interface that functions as a connection between the local and global buses, also incurring a two-cycle penalty).

4.3 Performance results

As with the previous experiments, the benchmarks that we used were drawn from the SPEC95 suite [26]. We used the `test` input set in all cases, although we reduced the number of iterations in some programs (after performing experiments to ensure that the reduced number of iterations did not perturb our results).

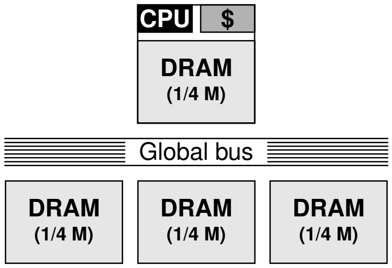
We simulated six of the SPEC95 benchmarks: `go`, `mgrid`, `applu`, `compress`, `turb3d`, and `wave5`. We ran each benchmark for 200 million instructions or to completion, whichever came first. We did not statically replicate any data pages; all pages were distributed round-robin across all nodes. We ran simulations for both two-processor and four-processor DataScalar systems. Each processor has sufficient capacity to hold one-half and one-fourth of the data set, respectively, for each benchmark. Our target DataScalar system dynamically replicates remote data in the data cache, as described in Section 4.1.

We compared the Datascalar performance against two points: an identical processor with a perfect data cache (single-cycle access to any operand), and a more traditional system which has the same amount of on-chip memory as does one chip in each DataScalar experiment. We thus compare a two-processor DataScalar execution against a system which has the same processor, half the memory on-chip, and half off-chip (to make a fair comparison, the buses are the same, and both systems cache updates at instruction commit, not issue). We show an example of this comparison, assuming four processors, in Figure 6. A traditional system (Figure 6a) being compared against a four-processor DataScalar machine (Figure 6b) would thus have one-fourth of its main memory on-chip and three-fourths off-chip.

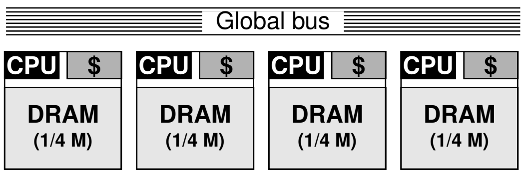
While the traditional system would certainly benefit if all of the on-chip memory was devoted to a large second- or third-level cache, measuring such a system against our simulated DataScalar implementation would be an unfair comparison. We consider the IRAM chip used in both types of system to be a commodity part, intended for stand-alone use, with the large on-chip memory functioning as main memory and not a cache.

In Figure 7 we plot the instructions per cycle for each experiment. We ran each benchmark assuming two and four DataScalar processors. The actual IPC value resides atop each bar. We see that the performance benefits that the DataScalar system has to offer are substantial. The results are particularly striking for `compress`, almost a doubling of IPC over the traditional architecture. That performance gain is so much higher because `compress` issues almost as many stores as loads, which never have to go off-chip in a DataScalar system. For all other benchmarks, the DataScalar system manages to capture much of the available ILP, approaching the IPC of the perfect data cache in some cases.

The DataScalar system deals with a finer-grain distribution of memory better than does the traditional system; the drops in DataScalar performance when going from two-processor to four-processor systems are less than 0.05 IPC (the comparable drops in



(a) More traditional architecture



(b) DataScalar architecture (4 nodes)

Figure 6. Comparing two IRAM organizations

performance on the traditional system range from 0.1 to 0.6 IPC). The IPC for `wave` actually improves when running on four processors instead of two (the benefits of more processors running datathreads concurrently outweigh the additional off-chip communication). In only two cases (`mgrid` and `turb3d` with two nodes) does the DataScalar system perform worse than the traditional system. This abnormality results from poor correspondence protocol performance (a high rate of false hits at one node causes the other node to stall frequently, waiting for the owner to commit the offending load and issue a reparative broadcast).

We present the results of a sensitivity analysis in Figure 8. The two benchmarks presented are `go` and `compress`, each of which was run to completion. For each benchmark, we plot results assuming the same parameters that we used for the experiments in Figure 7, except that we vary one parameter in each graph. The parameters we varied were: data cache size, main memory access time, global bus clock speed, width of the global bus, and number of RUU entries. On each graph, we plot the IPC for the same five systems as we measured in Figure 7 (perfect data cache, two- and four-processor DataScalar machines, and traditional systems assuming one-half and one-fourth of the main memory on-chip).

We see that the DataScalar runs consistently outperform the traditional runs over a wide range of parameters. As expected, the performance of the two types of systems converges when memory bank access times come to dominate the latency of a memory request (because DataScalar systems reduce the overhead of transmitting the data, not accessing them). Conversely, when the speed differential between the global and on-chip buses grows, so does the disparity between DataScalar and traditional performance.

In Table 3 we list a few of the BSHR and broadcast statistics from the two-processor runs. We list the percentage of broadcasts that were issued late, at commit time, due to false hits. These percentages will drop for larger caches, since the probability that a block will be replaced in between issue time and commit time is inversely proportional to cache size. The middle column lists the percentage of BSHR entries that were squashed due to false hits. The right-most column lists the percentage of remote accesses that were waiting in the BSHR for the local processor's request. Those values range from 2% to 9%, showing evidence that some effective datathreading is occurring, since a processor needs to be running fairly far ahead of another for that situation to occur.

4.4 Other implementation issues

In this subsection we describe other issues pertinent to DataScalar architectures—namely, the cost overheads of the extra processors, the expense of requiring broadcasts, speculation in DataScalar, and operating systems issues.

Cost

Conventional systems today typically consist of a single processor and a collection of memory chips. Each of these components comprise a significant fraction of the total cost of the system. 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. When comparing the cost of a DataScalar system and a traditional system with one processor and "dumb memory" (such as the comparison in Figure 6), the DataScalar system becomes cost-effective when the performance it adds outstrips the cost of the additional processors.

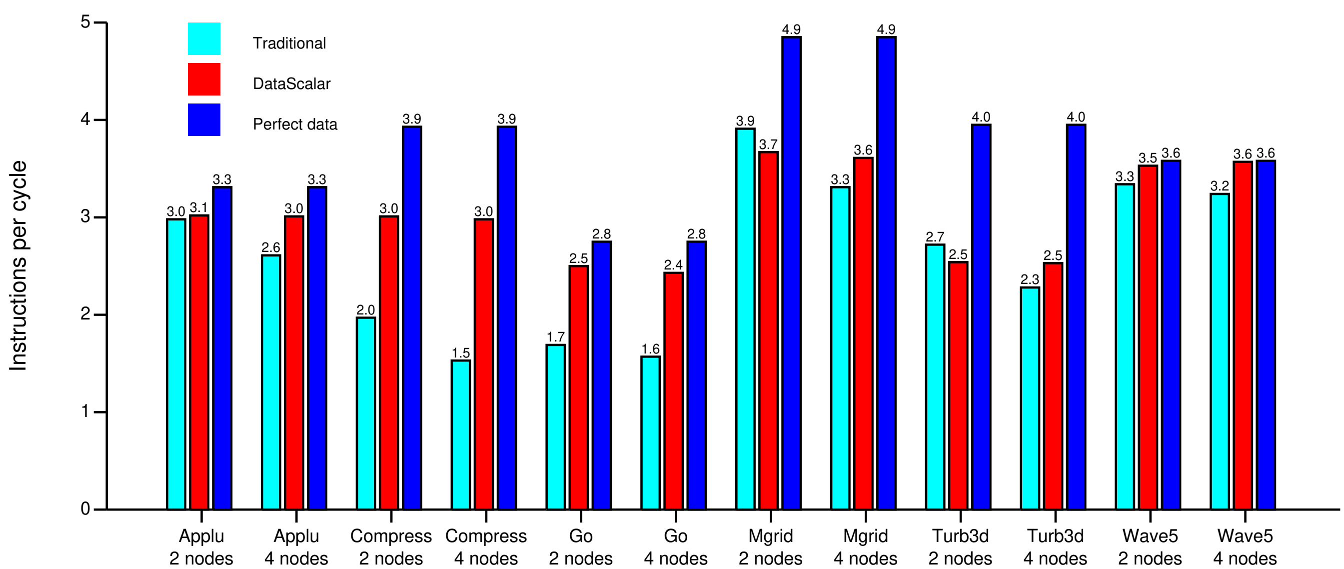


Figure 7. Timing simulation DataScalar results

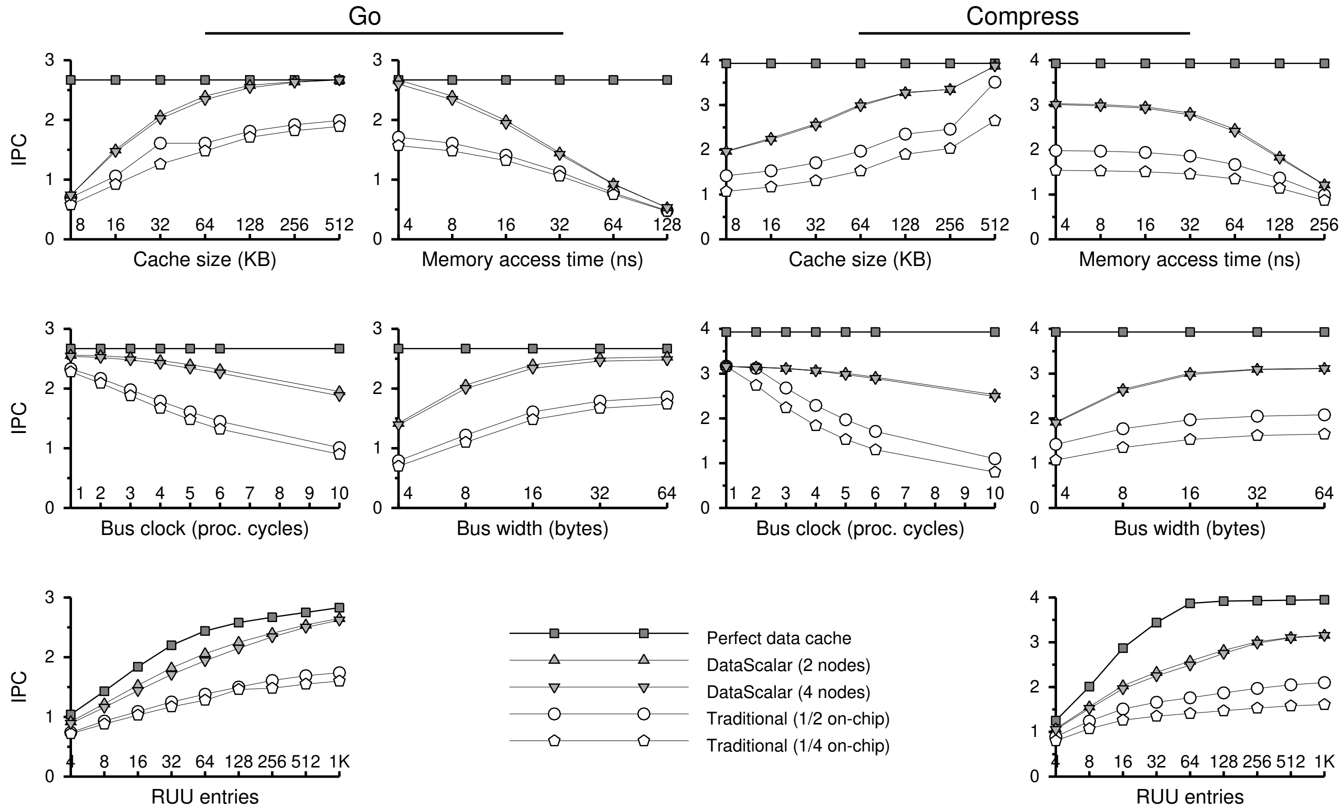


Figure 8. Sensitivity analysis of DataScalar experiments

Benchmark	Late broadcasts		BSHR squashes		Data found in BSHR	
(# of nodes)	2	4	2	4	2	4
`applu`	10%	9%	12%	12%	10%	7%
`compress`	11%	8%	16%	22%	8%	4%
`go`	9%	10%	12%	15%	19%	7%
`mgrid`	23%	21%	31%	31%	6%	4%
`turb3d`	38%	37%	59%	59%	3%	1%
`wave5`	9%	7%	11%	3%	3%	1%

Table 3: DataScalar broadcast statistics

The parameters are the same as for the experiments described in Section 4.3. The numbers are the arithmetic mean at all nodes. The percentages are out of total number of broadcasts (column one) and out of BSHR accesses (columns two and three).

Wood and Hill showed [28] that for a parallel system to be cost-effective, the *costup* (the relative increase in total cost as more processors are added) should be less than the *speedup* (the relative increase in performance as more processors are added). When memory or interconnect costs dominate those of the additional processors, the system may still be cost-effective even if the speedups are comparatively small.

A majority of the die of most modern processors is devoted to memory, even though the total cache capacity for each is generally only in the tens of kilobytes. We believe that the ratio of on-chip memory area to total chip area will continue to grow in the future, making the relative expense of the processing logic shrink over time. If true, this trend will make memory and packaging the dominant costs of future systems. DataScalar architectures could thus be cost-effective, even though the speedups they provide are much less than linear.

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 both expensive and not scalable. On certain interconnects—such as on a ring or bus—they may be effected with only minor additional cost, though reliable delivery and error recovery are inevitably more complicated for broadcast operations.

Broadcasts on a bus are free, since every bus transaction is an implicit broadcast. However, the very feature that makes broadcasts cheap—the centralized nature of a bus—makes the bus an unlikely candidate for the high-performance interconnect of the future. However, the demise of the bus has been much slower than predicted, and buses may persist for some time to come.

Ring operations, such as the IEEE/ANSI standard Scalable Coherent Interface [16, 23] seem well-suited for this kind of operation. On a ring, operations are observed by all nodes if the sender is responsible for removing its own message. We envision a ring interconnect because of the high-performance capability [22], but broadcast on a ring is complicated by the fact that operands originating at different processors are received at other nodes in different orders. A simple tag can sort out data to different addresses, but the issue is complicated when two accesses to the same datum are broadcast close in time. Complications also arise whenever certain data items must be rebroadcast (e.g., because a receive queue is full), or cancelled.

One technology that may be an excellent match for DataScalar programs running on large systems is optical interconnects. One of the properties of free-space optical interconnects is that they have extremely cheap (essentially free) broadcasts. For massively parallel systems that use optical interconnects, the SPSD execution model may be a good way to reduce the execution time spent in serialized code, thus improving scalability [2].

Speculative execution

Fine-grain speculative execution is now present in state-of-the-art processors, and a successful DataScalar architecture must be compatible with speculation. Much of the promise we see in DataScalar comes from out-of-order execution, which enables multiple processors 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 frequent superfluous broadcasts would greatly hinder performance. The two endpoints for speculative policies are (1) to hold onto speculative broadcasts until the speculative condition is resolved, and (2) to send the broadcast immediately upon issue and then send a corresponding squash if the load that generated the broadcast is squashed.[1] The former conserves bandwidth at the expense of added latency, while the latter consumes bandwidth while reducing latency (bandwidth limitations add latency, however, so there is likely a crossover point at which one policy becomes better than the other). A promising approach is to assign confidence values to speculative loads; loads with high correctness confidence should be broadcast and squashed if incorrect, whereas loads with low confidence should be held locally until the speculative condition is resolved.

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 at different processors, but would cause no special problems. However, asynchronous events could potentially cause difficulty if they are not observed at precisely the same point by all processors. Consider the case in which a write causes a page fault. Since only one processor actually performs a write to communicated data—the other processors all simply discard their result—only the owning processor would observe the page fault. If the other processors 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 processors have the same page table entries, and actually check for exceptions on every memory operation. Thus each processor 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.

5 Exploiting parallelism with DataScalar

DataScalar is a memory system optimization, intended for codes that are (1) limited by the memory system and (2) difficult to parallelize. Every DataScalar machine is a *de facto* multiprocessor, however. When codes contain coarse-grain parallelism, the system should be run more like a traditional multiprocessor. We view DataScalar and multiprocessing as two endpoints on a spectrum: the former runs transparently and makes no attempt to exploit parallelism in the code; the latter requires compiler and/or programmer support, and its main focus is explicit exploitation of coarse-grain parallelism.

In this section we discuss some of the intermediate points along this spectrum. In Section 5.1 we describe new opportunities for exploiting medium-grain parallelism in a DataScalar architecture. In Section 5.2 we present some issues associated with running a mixed-mode program, in which some parts of the program run in DataScalar mode and others run as on a traditional MIMD machine.

5.1 Multiprocessing from a Datascalar perspective

DataScalar systems benefit from both ESP and datathreading transparently, without requiring recompilation. In many codes there is limited parallelism, which is insufficient to justify porting the code to a full-blown parallel processor, but which could improve DataScalar performance. We propose using software support to expose this parallelism in a DataScalar context.

When the programmer or compiler identifies an isolated block of code that can be executed on one node and not the others (requiring few or no remote operands), that processor may temporarily "peel off" from ESP mode. The processor will then compute some result, and either store the result locally, or communicate it to the other processors when the processor rejoins ESP execution. We call this technique *result communication*. Only the participating processor(s) must execute the code block in question; the others must branch around the code. The compiler must perform the analysis to ensure that the non-participating nodes do not need any of the intermediate results generated during the isolated computation.

Result communication has the potential to reduce global traffic as well as reduce the critical path at the non-participating nodes. We note that this is a special case of data parallel execution; if all nodes are independently performing computations and writing the results to communicated data that they own, the execution looks similar to traditional data parallel [5].

Performing this parallelizing analysis in a DataScalar context has distinct advantages. Hardware support in the processor, coupled with run-time system support (described below), can assist in making run-time checking efficient. When the data for a computation are scattered across all nodes, the system has an efficient fall-back case (asynchronous ESP). The compiler's job is therefore to provide the hardware with *options* (break from ESP *if* most or all of the data are local), not *guarantees* about where the data are located.

When a processor deviates from ESP, the code it executes must not perturb the state of the correspondent caches, if the caches are indeed used to implement dynamic replication. We may prevent such a perturbation by marking non-ESP accesses, and dropping them at commit time, rather than updating the caches. If, while in this mode, a processor requires a communicated datum that it does not own, either the compiler must have generated code to force the owning processor to send the operand, or the processor could fall back to a traditional request/response model. The most important guarantee the compiler must make, however, is that there are no side-effects in the isolated code that were supposed to have changed the state at every processor.

In the rest of this subsection we present an illustrative example of result communication. The branch test could be implemented by performing a test on a bit stored in the page table. We call this test the "local" function. A processor might take the branch if the `local` bit in the TLB for a given address was zero (the `local` bit corresponds to ownership of the operand). To make this run-time check efficient, however, the processors should not have to check ownership of multiple addresses, the resolution of some of which the processors would need to run the code being parallelized to resolve. We can fold the check for multiple operands into a single check, by securing a guarantee from the run-time system that certain data items are placed local to the same processor. One technique for doing so is to modify the run-time storage allocator by both giving it an ownership assignation function and passing it an address on a request for storage. The storage allocator then ensures

1. A squash is necessary because all incoming broadcasts are buffered in the BSHRs. Incorrectly speculated broadcasts would not always be matched with a request from the receiving processor, and would thus accumulate until the BSHR was full. An alternative solution is to periodically flush the BSHRs, cleansing them of stale broadcasts.

that the allocated storage was allocated at the same processor as the address that was passed to it. The goal is to ensure that all of an aggregate (either an array or a dynamically allocated structure) falls entirely on one node. Binding the aggregate to one node should be avoided if the aggregate is so large that it should be spread across multiple nodes.

The notion of placing data can be generalized so that pages containing parts of a structure are tagged with a class identifier. This would decouple structures from specific nodes, so that the number of processors (and page assignation) could change dynamically (the OS would ensure that all of the pages of a given class came to reside entirely on one processor).

In Figure 9 we depict an example that could make effective use of result communication. We show a chained hash table, the main array of which is distributed across multiple processors. Below the hash table we show the high-level code transformation that would alter the insertion routine. The additions to the code are marked with arrows. Whenever a collision occurs and an entry must be added to a chain, the run-time allocator (`malloc` in this example) is passed the address of the head of the chain, and it returns storage that is local to the same processor as is the head of the chain. When insertions or deletions into the hash table need to be made, the compiler places a branch around the insertion or deletion code. The processor that owns the head of the chain will thus own the entire chain, and can make the insertion or deletion without requiring any off-chip communication.

Since every part of the array resides local to some processor, *every* insertion and deletion can be made with *no remote communication*. The run-time test is efficient because the locality bit for the head of the table is obtained with a simple address translation, and the tests for every element in entire chain are subsumed by the locality test for the head of the list. Many such optimizations are possible, and are a promising area of research.

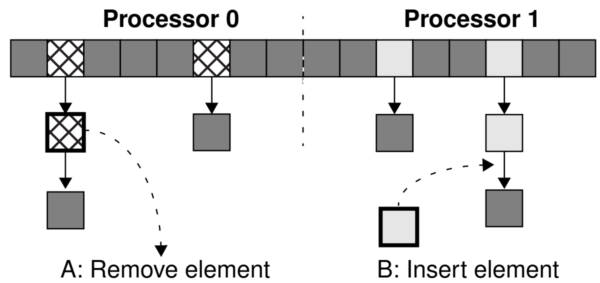
5.2 DataScalar from a multiprocessing perspective

If much coarse-grain parallelism is extractable from a program, the program should be run using the system as a multiprocessor rather than as a DataScalar architecture. Few codes are "embarrassingly parallel," however. Codes should perhaps be run on a hybrid architecture, which runs a program as a multiprocessor when there is sufficient coarse-grain parallelism, and as a DataScalar architecture when there is not (thereby reducing serial overheads). The program may either switch statically or dynamically between the two modes, or run both modes simultaneously, using tagged instructions or regions of data addresses to decide which mode to use at a given point.

Future microprocessors will soon have sufficient resources to put multiple processing units on a single chip. Whether chip multiprocessors (CMPs) will succeed is a subject of debate in the research community. Widespread use of CMPs is more likely if techniques are developed that speed up uniprocessor programs—particularly those that are not easily parallelizable—by running them on a tightly coupled multiprocessor. DataScalar architectures may be one such candidate, depending on the future disparity between the cost of near versus far intra-chip communication.

Given the wide variance of characteristics across important applications, CMPs may benefit by supporting multiple modes of execution, so as to execute the broadest possible range of applications efficiently. To this end, we propose a "three C's" model for CMP execution modes.[1] In our model, we characterize an application as being limited by either *computation*, *control*, or *communication*. We propose to run a CMP as a different organization for each limiting category.

1. Our model is not to be confused with Mark Hill's "three C's" model that characterizes cache misses [14].




```
struct t_entry *HASH_TABLE[SIZE];
insert_entry(int key)
{
  struct t_entry *entry;
  int h_index = HASH_FN(key);
  entry = (struct t_entry *)
          malloc(sizeof(struct t_entry),
                 &HASH_TABLE[h_index]);
  if (local(&HASH_TABLE[h_index])) {
    entry->key = key;
    entry->next = HASH_TABLE[h_index];
    HASH_TABLE[h_index] = entry;
  }
}
```

Figure 9. Medium-grain parallelism in a hash table

- **Computation-bound**: the application has enough coarse-grain parallelism that functional unit throughput is the limiting factor. If enough work can be found to allow all processing units to be utilized, the CMP should be run as a conventional multiprocessor.
- **Control-bound**: limited instruction-level parallelism requires the CMP to perform coarse-grain speculation. Control dependencies prevent a single processor from running very far ahead. The program execution could therefore benefit from having the processors function as stages in a Multiscalar-like architecture [12, 25], wherein each processor speculatively executes large blocks of code. The processor thereby obtains a much larger instruction window in which to find sufficient ILP.
- **Communication-bound**: a single processor is limited by frequent, slow accesses to non-local (on-chip) memory banks. The program execution could benefit from an architecture that uses multiple processors to reduce communication overheads. DataScalar architectures are one such example, and may be a good match for this category.

Many applications are unlikely to fall squarely into one category or another; it is probable that an application will run alternately in two or all three of these modes. Both characterizing applications in this manner, and designing the mechanisms needed to support concurrent execution of multiple modes, are interesting research issues to be explored.

6 Future directions and conclusions

This work began as a solution for running programs on multiple single-chip computers, when the physical memory requirements exceeded the capacity available on a single chip. Other groups are also working on issues associated with the integration of processors and main memory, notably the IRAM project at UC-Berkeley [20] and the PPRAM project at Kyushu University [18]. Some companies are already prototyping preliminary designs that incorporate processing logic with a substantial amount of memory on a single die; Sun Microsystems [21] and Mitsubishi [9] are two examples. Our proposal uses multiple processing units to enhance the performance of uniprocessor programs. Other groups are also exploiting this concept—Stanford Hydra [19] uses automatic parallelization, Wisconsin Multiscalar [12, 25] uses coarse-grain speculation, and both the M-machine [10] and the Simultaneous Multithreading work [27] use aggressive multithreading.

In this paper we have proposed a system organization that exploits cheap computation to reduce memory system overheads. DataScalar architectures combine dynamic execution and speculation with the Massive Memory Machine's ESP execution mode. This class of architectures works well when remote memory accesses are significantly more expensive than local memory accesses, when global broadcasts are relatively inexpensive, and when the cost of additional processors is a small addition to the total cost of the system. We evaluated one conceivable DataScalar implementation, and showed that these ideas can indeed improve performance when memory system performance is critical.

Our current research efforts include a performance decomposition of DataScalar execution, to determine the relative fractions of performance gains coming from the elimination of write traffic, the elimination of requests, and datathreading. We are extending our cache correspondence protocol to handle speculation. We are also studying the potential of static replication, both coarse- and fine-grained, the effectiveness of more coarse-grained dynamic replication, and software support for improving datathread length. Finally, a major effort underway addresses exploiting parallelism within the DataScalar context, both at a medium grain (compiler support for result communication) and at a coarser grain (mixed-mode execution of SPSD and MIMD).

Acknowledgments

The authors thank Alain Kägi, Scott Breach, Babak Falsafi, Mark Hill, Milo Martin, Steve Reinhardt, Mike Smith, T.N. Vijaykumar, David Wood, and Todd Bezenek for their helpful discussions and comments on drafts of this paper. We also thank Todd Austin, who developed the original SimpleScalar tool set.

References

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

[2] Doug Burger and James R. Goodman. Exploiting Optical Interconnects to Eliminate Serial Bottlenecks. In *Proceedings of the Third International Conference on Massively Parallel Processing Using Optical Interconnects*, October 1996.

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

[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] 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.

[6] Don Draper, Jeff Yetter, Ahsan Bootehsaz, Michael A. Buckley, Charlie X. Huang, Yusuke Ohtomo, Jurij Paraszczak, and Donald A. Priore. Panel Discussion on the Interconnect Nightmare. In *Proceedings of the 1996 International Solid-State Circuits Conference*, pages 278–279, February 1996.

[7] J. H. Yoo et al. A 32-bank 1Gb DRAM with 1 GB/s Bandwidth. In *Proceedings of the 1996 International Solid-State Circuits Conference*, pages 378–379. Samsung Electronics Co., February 1996.

[8] Masashi Horiguchi et al. An Experimental 220MHz 1Gb DRAM. In *Proceedings of the 1995 International Solid-State Circuits Conference*, pages 252–253. Hitachi, February 1995.

[9] 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.

[10] Marco Fillo, Stephen W. Keckler, William J. Dally, Nicholas P. Carter, Andrew Chang, Yevgeny Gurevich, and Whay S. Lee. The M-Machine Multicomputer. In *Proceedings of the 28th International Symposium on Microarchitecture*, pages 146–156, November 1996.

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

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

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

[14] Mark D. Hill. *Aspects of Cache Memory and Instruction Buffer Performance*. Ph.D. thesis, University of California at Berkeley, November 1987.

[15] Liviu Iftode, Kai Li, and Karin Petersen. Memory Servers for Multicomputers. In *Proceedings of the 38thIEEE Computer Society International Conference (COMPCON)*, pages 538–547, February 1993.

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

[17] Osamu Kimura, Richard Crisp, Michael Nagy, Henry Lie, Roelof Salters, Kenji Numata, Takao Watanabe, and Kazunori Saitoh. Panel Session: DRAM + Logic Integration: Which Architecture and Fabrication Process. In *Proceedings of the 1997 International Solid-State Circuits Conference*, February 1997.

[18] Kazuaki Murakami, Satoru Shirakawa, and Hiroshi Miyajima. Parallel Processing RAM Chip with 256Mb DRAM and Quad Processors. In *Proceedings of the 1997 International Solid-State Circuits Conference*, pages 228–229, February 1997.

[19] Basm A. Nayfeh, Lance Hammond, and Kunle Olukotun. Evaluation of Design Alternatives for a Multiprocessor Microprocessor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

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

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

[22] 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.

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

[24] 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.

[25] Gurindar S. 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.

[26] Standard Performance Evaluation Corporation. *SPEC Newsletter*, Fairfax, VA, September 1995.

[27] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 1995.

[28] David A. Wood and Mark D. Hill. Cost-Effective Parallel Computing. *IEEE Computer*, 28(2):69–72, February 1995.