Journal of Systems Architecture 45 (1999) 1001-1022, Elsevier Science B.V.

DataScalar: A Memory-Centric Approach to Computing

Stefanos Kaxiras, Doug Burger, James R. Goodman
University of Wisconsin-Madison
{kaxiras,dburger,goodman}@cs.wisc.edu
Computer Sciences
1210 W. Dayton St.
Madison Wisconsin 53705 U.S.A.

Abstract—*Commodity microprocessors contain more on-chip memory with each successive generation, and will contain tens of megabytes within the decade. We describe a novel architecture that runs an unmodified uniprocessor program across multiple nodes, each of which contains a processor tightly integrated with a sizable memory. The execution of instructions is replicated, while the access of operands is distributed across the nodes. Each node accesses operands in its fast local memory and broadcasts them to the other nodes. This architecture exploits out-of-order execution and the fact that each chip has integrated processor and memory, to run memory-intensive, hard-to-parallelize programs more efficiently. In this paper, we describe an implementation with specific solutions to the unique problems that this architecture poses. Finally, we conclude by comparing simulation results of our implementation to more traditional equivalent systems. In our simulated implementation, five unmodified SPEC95 binaries ran —in most cases— considerably faster than in systems with more traditional memory systems.*

Keywords: Memory Architecture, Processor-Memory Integration

1 Introduction

For a long time, computer performance has been increasing at an exponential rate. A major contributor to this sustained progress has been an increase in the density of packaging. In the last twenty years, the focus has been on ever more sophisticated processors, but an increasingly large proportion of the processor has been devoted to memory. As the speed gap between off-chip memory and the processor widened, more and more sophistication has appeared, today providing multiple levels of support independently for instructions and data. This trend is likely to continue, with much current interest in memory/processor integration, *i.e.*, placing increasingly large portions of the memory system on the same chip or module with the processor. Some recently announced microprocessors have as much as 80% of their transistors devoted to memory.

As performance levels continue to grow, the speed gap between on-chip processing and off-chip memory access continues to widen, suggesting that progressively more expensive and sophisticated techniques are required to support the conventional off-chip memory system. Many techniques are being pursued toward the goal of reducing or tolerating the latency of off-chip memory accesses. Even with aggressive memory hierarchies, modern processors spend much of their time waiting for needed operands, both instructions and data [2]. Given the continuing increases in processor speed and of main memory sizes, increasing the disparity between processor and memory cycle times, it seems unlikely in the long term that these approaches will bridge the gap. Memory latency tolerance/reduction techniques — such as non-blocking caches [15,7], hardware and software prefetching [4,6,5,9,14,17], multithreading [16,23], and out-of-order execution [28,25] — may reduce memory-related processor stalls until available memory bandwidth is saturated. A recent study showed that processors that are aggressively latency-tolerant owe fully half of their memory stall times to limited off-chip bandwidth [2]. Unfortunately, many of the techniques employed to reduce or tolerate latency are only deferring a pending crisis by increasing the bandwidth requirements of the processor's pins. Techniques such as out-of-order execution pose increasingly severe demands on the processor's pins, if for no other reason than that they succeed in speeding up the processor. Speculative execution and prefetching run the risk of severely increasing bandwidth requirements.

While the processor may no longer be stalled for extended periods waiting for off-chip cache misses, the memory limitation poses a serious barrier for applications that are difficult to parallelize. Thus it would be highly desirable to find ways to extend the range of usefulness for uniprocessor programs beyond the point of a single-chip computer, or IRAM, even as the available memory within the IRAM grows.

We divide application programs (or perhaps phases of application programs) into two categories:

- The program's data set fits comfortably in on-chip memory (DRAM) achieving satisfactory performance.

- The program’s data set is too large for the on-chip main memory, with the result that the processor experiences thrashing.

Programs in the second category may benefit from conventional multiprocessing, using multiple nodes to achieve either shared-memory or message-based parallel computing. If the program can be easily parallelized the collection of IRAM nodes can be used a conventional DSM multiprocessor [20]. That leaves the problem of programs that are too large to fit within a single node, but are not-easily parallelized. For such applications, we describe a novel approach, first proposed in [3], that can retain the uniprocessor programming model that we believe can achieve higher performance than a conventional uniprocessor. A program’s data set is spread across these nodes. All processors run the same program, broadcasting operands they own to the other processors when needed, and performing any tasks that can be accomplished entirely on-chip without off-chip communication.

The rest of this paper unfolds as follows: In Section 2, we describe the DataScalar proposal, enumerating three major advantages it has over traditional architectures, and describing how each advantage improves performance. We also discuss implementation issues associated with these types of systems. In Section 3, we present a performance evaluation of DataScalar. In Section 4 we discuss *result communication*, an extension to the basic architecture to enhance its performance. Finally, in Section 5, we list other research efforts related to processor/memory integration, present future directions, and conclude.

2 DataScalar architectures

The DataScalar architecture is intended to address the dual constraints of (1) single-chip computers with hard limits on memory expansion and (2) existing—or only slightly modified—uniprocessor programs that are not easily parallelized. We propose to exploit the availability of multiple processors to minimize memory latency by exploiting the fact that all memory is local to *some* processor. Thus each read operand can be quickly fetched by some processor, and each memory update can be achieved by means of a local write by some processor. Communication consists of the information necessary for all processors to execute the entire program. 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).

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 [10]. 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).

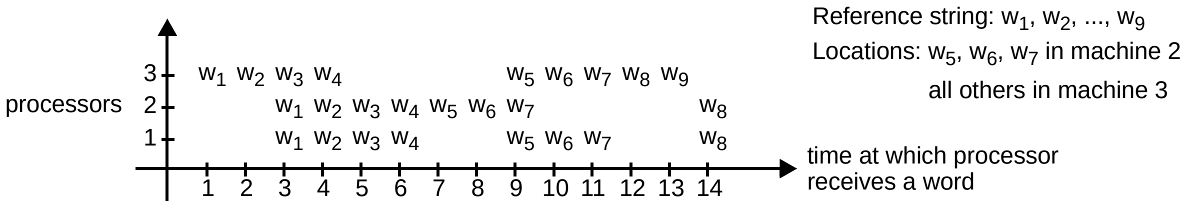

Reference string: w_1, w_2, ..., w_9
Locations: w_5, w_6, w_7 in machine 2
all others in machine 3
processors
3
2
1
w_1 w_2 w_3 w_4 w_5 w_6 w_7 w_8 w_9
w_1 w_2 w_3 w_4 w_5 w_6 w_7 w_8
w_1 w_2 w_3 w_4 w_5 w_6 w_7 w_8
1 2 3 4 5 6 7 8 9 10 11 12 13 14
time at which processor receives a word

FIGURE 1. MMM

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 4.

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). 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. The address space is thus divided into two categories: *replicated* and *communicated* data. Replicated data are present in each node and communicated data are distributed among the nodes. Dynamic replication of data introduces some new consistency issues that we will discuss in Section 2.4.

Figure 2 shows how loads and stores differ for replicated and communicated data. In this figure data *A* through *P* are distributed among the four nodes (communicated data) while datum *Q* is replicated in every node (replicated datum). Loads and stores to *Q* complete locally in every node without any communication. A load to communicated data is broadcast by the owner to the other nodes (e.g., node 0 loads and broadcasts *C* which becomes replicated if it is cached in all nodes). However, a store to communicated data completes only in the owner node while it is nullified in all other nodes (e.g. a store to *O* completes only in node 3).

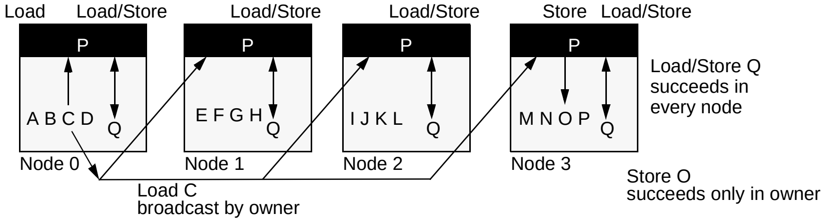


FIGURE 2. Only loads for communicated data are broadcast. Stores to communicated data are performed only by owner. Loads and stores to replicated data are performed by all nodes.

2.1 Push-mode operation

DataScalar systems enjoy the same benefits from ESP as did the MMM proposal: (1) reduced remote access latency, (2) elimination of request traffic, and (3) elimination of write traffic. Because each node runs the same program, a communicated operand can be sent to the other nodes 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 nodes, eliminating half of the communication delay by requiring only one-way communication. This "*Push-mode*," 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 on replicated data complete locally on every node. Stores or write-backs to communicated data occur only on the owning node, which preserves consistency since that node holds the only copy in main memory. Note that there are no consistency issues, because every node runs the same program.

2.2 Traffic reduction via ESP

The ESP model eliminates off-chip request traffic and off-chip write traffic. In this subsection, we measure the amount by which ESP reduces traffic to main memory. We assumed a 64-Kbyte, two-way set-associative, unified, write-allocate, write-back on-chip cache (this size is consistent with typical cache sizes at the time that SPEC95 [27] was written). 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 (megabytes) remaining, and the total number of distinct messages remaining (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 that spend much of their time stalled due to limited memory bandwidth—implementing ESP is likely to improve performance, or reduce the required system cost to achieve the same performance. Although these results focus on traffic reduction, they do not address the performance penalty associated with requiring broadcasts. We address that issue in Section 2.6.

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

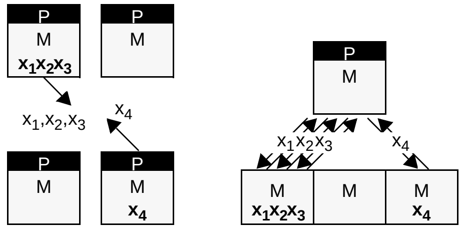
Table 1: Off-chip data traffic reduced by ESP

2.3 Datathreading

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 node contains a quarter of the program's physical memory. Figure 3b shows a more traditional organization, in which one node 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 operands happened to reside in the on-chip quarter of the memory, as opposed to a minimum of one for a DataScalar system.



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

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

FIGURE 3. Datathreading

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 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.

Experimental results [3] 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 instructions locally, and sending them to the other processors early—sometimes even before the other processors have resolved those addresses.

2.4 Dynamic replication and cache correspondence

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.

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. It is also possible to use lower levels of a multi-level cache hierarchy to perform dynamic replication. However, we use only the level-one caches because our particular solution requires a tight coupling of the cache and the processor. 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.

Forcing all loads and stores to execute in program order and after all previous (in program order) instructions have completed would ensure that caches are submitted to the exact same reference stream. However, modern architectures rely heavily on out-of-order instruction issue for their performance. We therefore allow loads and stores to issue out-of-order. We allow them to *peek* at the cache at issue-time but not change it. We use separate buffers to bring in data or to store new values, until the instructions commit at which point we update the cache.

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 remembering 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.

In Figure 4 we show how a series of load/store instructions can experience different hit/miss behavior. In this figure, two addresses, **X** and **Y**, conflict in the cache (which is depicted as different instances of a single cache-line). Instructions commit from left to right. If cache accesses were issued according to program order (Figure 4a) then $\mathbf{Y_1}$ would hit in the cache (assuming **Y** is already in the cache), $\mathbf{X_1}$ would miss, $\mathbf{X_2}$ would hit on the data brought by $\mathbf{X_1}$, and $\mathbf{Y_2}$ would miss since **Y** would be evicted by $\mathbf{X_1}$. However, the instructions may issue out of program order. One possible scenario is depicted in Figure 4b. In this situation, the second load to **X** ($\mathbf{X_2}$) misses when issued, (but would have hit at commit-time). This is an example of a *false miss*. Analogously, with a different issue order shown in Figure 4c, $\mathbf{Y_2}$ hits at issue-time because $\mathbf{Y_1}$ had just been committed, but should have missed at commit-time (e.g., if $\mathbf{Y_2}$ issued after $\mathbf{X_1}$ committed, it would cause a miss instead of a hit). We call this a *false hit*. False hits and false misses affect the reparative actions we must take in each node. The actions differ, depending on whether the node is the owner of the data or not.

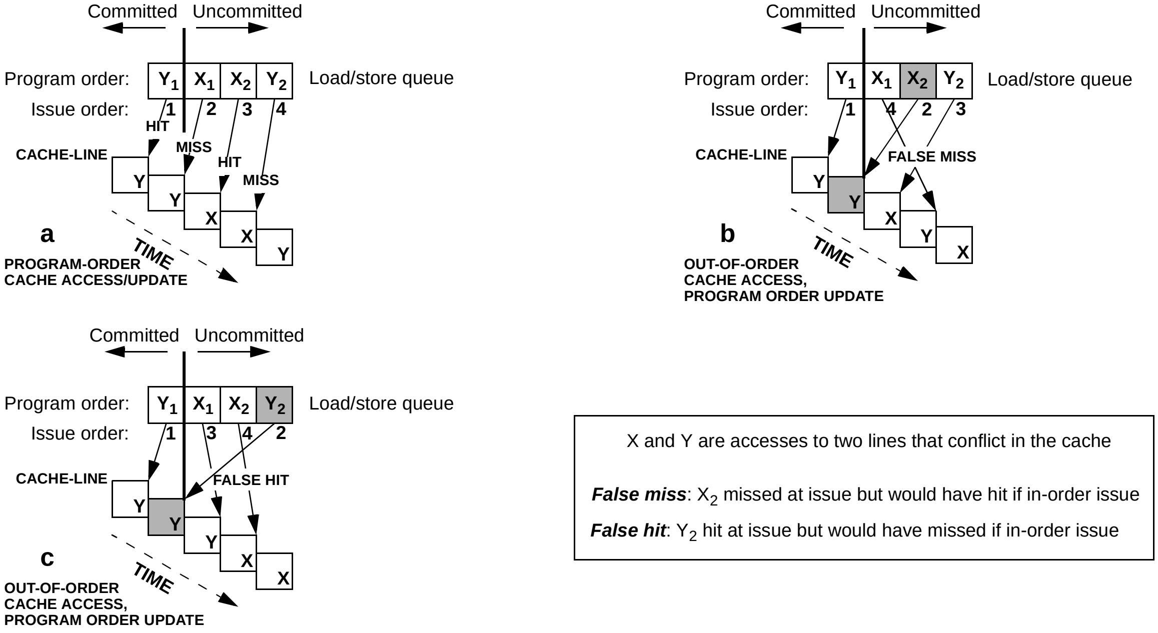


FIGURE 4. Program order and out-of-order cache access

In Tables 2 and 3 we show the actions that owner and non-owner nodes must take according to the combination of hits and misses that occur at issue-time and at commit-time. If the owner hits in the cache at issue-time (*True Hit* or *False Hit* cases in Table 2), it assumes that all nodes are also going to hit and does not broadcast anything. If however, at commit-time the hit proves to be a false hit (*False Hit* case in Table 2) the owner must broadcast the data. Non-owner nodes deal with false hits differently: if a node hits at issue-time (*True Hit* or *False Hit* cases in Table 3) it does not expect any broadcast from the owner. If however, at commit-time it misses (i.e., it had a false hit) it must consume the owner broadcast (which would otherwise occupy space in the node's queues).

Case	Issue	Commit	Action @ Issue	Action @ Commit	Comments
True Hit	HIT	HIT	—	—	
False Hit	HIT	MISS	—	Broadcast	False hit did not broadcast
False Miss	MISS	HIT	Broadcast if first to miss	—	False miss: One instruction of a **SEQUENCE** broadcasts
True Miss	MISS	MISS	Broadcast	—	

Table 2: Actions taken at issue and commit-time by owner node

Case	Issue	Commit	Action @ Issue	Action @ Commit	Comments
True Hit	HIT	HIT	—	—	
False Hit	HIT	MISS	—	Consume Broadcast	False hit failed to consume broadcast by owner
False Miss	MISS	HIT	Consume Broadcast if first to miss	—	False miss: One instruction of a **SEQUENCE** consumes the broadcast.
True Miss	MISS	MISS	Consume Broadcast	—	

Table 3: Actions at issue and commit-time taken by a non-owner nodes

By far the most complex cases to reason about are the false misses for both owners and non-owners (*False Miss* cases in Table 2 and Table 3). This is because false misses are the result of interference of multiple load/store instructions that reference the same data *in the same instruction window.* The only way an instruction can miss at issue and then, magically, hit at commit, is for another intervening instruction to bring the data in the cache. We deal with false misses by treating the sequence of accesses to the same cache line as a single entity. Regardless of the issue order, we recognize only a single miss at issue-time (the first) for the whole sequence, and allow only a single issue-time broadcast by the owner and a single issue-time broadcast/consume by the non-owner nodes. For example in Figure 4b $\mathbf{X_1}$ and $\mathbf{X_2}$ must generate only one miss. If $\mathbf{X_1}$ issues after $\mathbf{X_2}$, we "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.

2.4.1 Implementation

We will present our implementation of the cache correspondence protocol using a model of an out-of-order superscalar processor. In Figure 5 we show a block diagram of our processor model. Instructions are fetched, dispatched, issued to the functional units, and complete by updating registers (write-back).

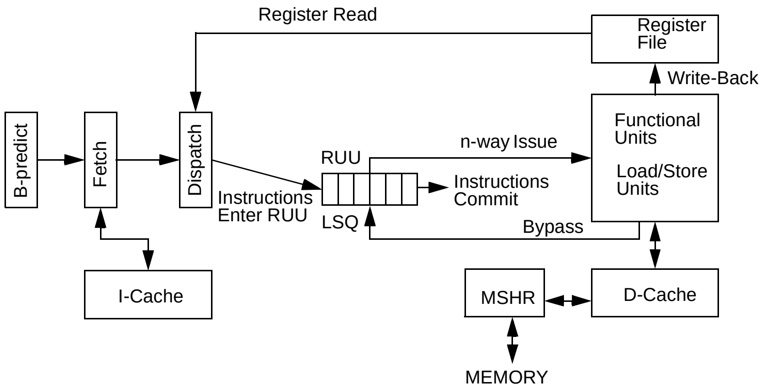


FIGURE 5. Out-of-order processor core

The operation of the out-of-order processor is based on Register Update Unit (RUU) that plays the dual role of reservation station pool and commit buffer [25]. We can envision the RUU as a FIFO queue (although an actual implementation would be a circular queue). Instructions enter the RUU from the left, issue to the functional units in any order and exit the RUU (i.e., *commit* or *retire* or *graduate*) in program order after their completion. An RUU entry contains the decoded instruction along with the available values of its arguments (i.e., register contents) and placeholders for the un-available arguments. A number of flags indicate the state of the entry. The functional and load/store units send their results to the register file and back to the RUU enabling waiting instructions to issue. For simplicity we assume that the Load/Store Queue (LSQ) is embedded in the RUU. The Load/Store Units access the L1 data cache. Multiple outstanding misses are handled using a pool of *Miss Handling Registers* (*MSHR*). A single MSHR is allocated for all misses on the same data block. However, it is de-allocated immediately after the miss is serviced from the lower level of the memory hierarchy.

We implement our cache correspondence protocol by expanding the basic out-of-order processor with a structure called *Commit Update Buffer* (CUB) (Figure 6). We envision separate CUBs for instructions and data, but in this paper we do not evaluate an instruction CUB. We consolidate the MSHR pool with the CUB in one structure to save space and unnecessary data copies. Potentially for any cache access, an entry is allocated in the CUB (Figure 6a). If the cache line is present in the cache, it is copied to the CUB/MSHR entry with a cache peek — an access that does not modify the cache in any way and especially its replacement bits (Figure 6b). If the cache peek results in a miss data are loaded from the lower level of the memory hierarchy directly into the CUB entry but not immediately into the cache. Since multiple memory operations on the same cache line are serviced by the same CUB entry, each RUU/ LSQ entry keeps a pointer to the corresponding CUB entry (Figure 6c). The CUB entries themselves keep a reference count. When an RUU/LSQ entry is committed, the cache tags are updated, and, if necessary, the cache line is copied from the corresponding CUB entry into the cache (Figure 6d). A CUB entry is not as short-lived as normal MSHRs: it is de-allocated only when its reference count drops to zero (i.e., when all the relevant RUU/LSQ entries have committed).

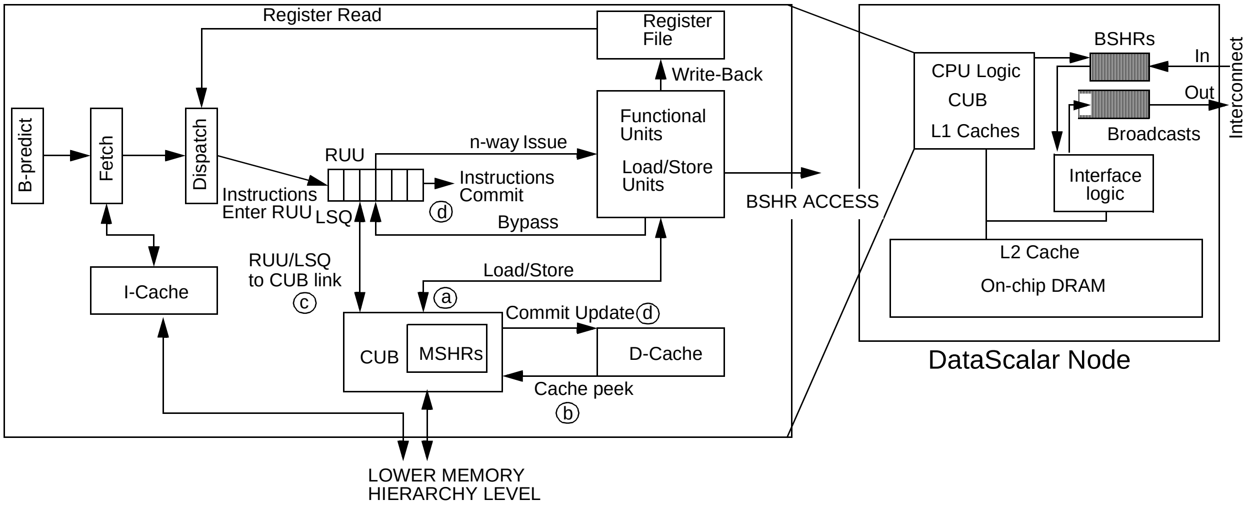


FIGURE 6. Commit Update Buffer (CUB) in a DataScalar node

Handling of false misses is simplified since a single CUB entry is used for any sequence of instructions that reference the same data block. The single CUB entry allows only the first true miss or false miss to broadcast the data or consume a broadcast. Subsequent misses at issue-time are serviced by the CUB entry. In addition to a pointer to the CUB entry, each entry in the RUU/LSQ is extended with state that represents whether the instruction missed in the primary cache at issue-time. This state is used at commit-time to determine whether the entry experienced a false miss, a false hit, a true hit, or a true miss.

2.4.2 Effects on performance

The CUB often increases performance of the out-of-order processor because it allows many load/store instructions to be resolved in the CUB/MSHR entries thus cutting down on cache traffic. It may also increase the hit ratio of the caches since it filters accesses. Both of these effects are the result of extending the life of the normal MSHRs by keeping them live as CUB entries until all instructions that reference them have committed. In Table 4 we present statistics for 6 of the SPEC95 benchmarks [27] when run on a single processor with and without a CUB. The CUB provides an increase in IPC ranging from 0% (`applu`) to 36% (`turb3d`). The hit rate of the L1 cache in four of the six benchmarks increases (up to 6% for `turb3d`) while for two benchmarks we have a small degradation (2% for `mgrid` and 1% for `applu`). The hit rate for the CUB entries (i.e., the number of times we find a cache line in the CUB) is higher than the corresponding MSHR hit rate for all benchmarks. This means that more instructions are serviced directly from the CUB and fewer accesses reach the L1 cache masking some of its misses. However, the CUB needs a higher number of CUB/MSHR entries to function efficiently, as it is evident from the average and maximum number of MSHRs that are used by each benchmark. Furthermore, the coupling of the MSHRs with the dynamic core must be very tight since multiple instructions need to access and retrieve data from the MSHR pool simultaneously.

	`go`		`mgrid`		`applu`		`turb3d`		`compress`		`wave5`	
	CUB	**—**	**CUB**	**—**	**CUB**	**—**	**CUB**	**—**	**CUB**	**—**	**CUB**	**—**
IPC	1.68	1.64	3.91	3.82	2.98	2.98	2.72	2.00	1.97	1.90	3.34	3.18
Cache Hit rate	0.96	0.94	0.90	0.92	0.88	0.89	0.86	0.80	0.78	0.77	0.96	0.94
MSHR Hit rate	0.35	0.30	0.55	0.43	0.28	0.24	0.39	0.26	0.08	0.03	0.34	0.23
Ave. # MSHRs	6.36	0.67	14.65	1.12	8.67	0.53	5.19	2.41	8.65	0.91	7.83	0.30
Max# MSHRs	54	20	24	18	23	15	18	9	21	10	17	6

Table 4: Performance statistics with and without a CUB

2.5 Speculative execution

Fine-grain speculative execution is now appearing in many state-of-the-art processors, and a successful DataScalar architectures 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 nodes 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 could spell disaster for this system. 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. 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.

2.6 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 networks, such as the IEEE standard Scalable Coherent Interface (SCI) [12,24] seem particularly well-suited for this kind of operation. On a ring, all 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 [21], but broadcast on a ring is complicated by the fact that operands originating at different nodes 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.

3 Performance evaluation

We evaluated a DataScalar system consisting of multiple integrated processor/memory (IRAM) modules connected via a global bus. In Figure 6 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 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 unowned by that processor. In Figure 6 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. Each page table entry has one bit that determines ownership of a page.

3.1 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) [25] 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 (LSQ), 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 [26]).

3.2 Memory system parameters

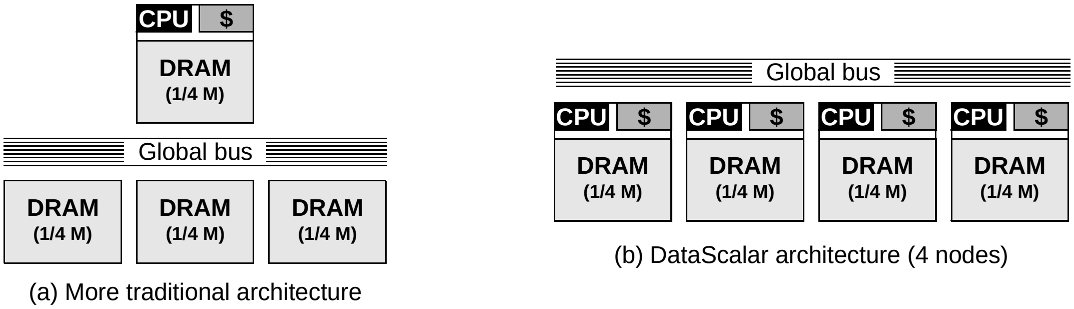
On-chip memories are likely to be significantly faster than DRAMs are today. Using sub-banking, with hierarchical word- and bit-lines, will enable DRAM banks to have access latencies that are comparable with those of cache memories. Current high-density (1 Gb) DRAM prototypes, the processes of which are optimized for density and not speed, have access latencies in the low 30's of nanoseconds [29,11]. 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).

3.3 Comparisons

We compared the Datascalar performance against three points: an identical processor with a perfect data cache (single-cycle access to any operand), and two more traditional systems which have 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 update the caches at instruction commit, not issue). We show an example of this comparison, assuming four processors, in Figure 7. A traditional system (Figure 7a) being compared against a four-processor DataScalar machine (Figure 7b) would thus have one-fourth of its main memory on-chip and three-fourths off-chip.



(a) More traditional architecture

(b) DataScalar architecture (4 nodes)

FIGURE 7. Comparison Systems

Two versions of the traditional system are examined:

1. Static allocation of memory on the on-chip DRAM: In this version we allocate statically a number of pages on the on-chip memory. Different policies to select pages are possible: (i) selecting a contiguous part of the memory, (ii) selecting the most heavily accessed pages as determined by profiling. The disadvantage of this scheme is that the static allocation of memory leads to suboptimal performance. Profiling can help considerably as long as it provides good feedback not specific to the profiling program input. However, this scheme is the simplest to implement and it does not incur run-time overhead. Another reason we emphasize this scheme is our belief that a commodity IRAM chip (used in both the DataScalar and the traditional architectures), intended for stand-alone use, would use its large on-chip memory as main memory and not as a cache.
2. Page cache: In the second version of the traditional system we treat the on-chip memory as a page cache. Pages (4 Kbytes) are brought on-chip on demand. The cache is fully associative and replacement is handled in software. The virtual memory system is modified to differentiate between on-chip and off-chip pages. A reference to an off-chip page is considered a “soft” page fault. An on-chip page is chosen for replacement (in software) and the referenced off-chip page is brought in. The disadvantages of this scheme include the replication of a huge part of the main memory in the on-chip cache and overhead for servicing soft page faults.

In Figure 8 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 with static memory allocation. 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 traditional system with a page cache suffers from limited off-chip bandwidth. Its performance is reasonable when the available on-chip memory is about half the program size (it outperforms static memory allocation for `go` and `turb3d`). However, when the available on-chip memory drops to about one quarter of the program size `compress` and `turb3d` experience severe page thrashing with devastating effects on performance.

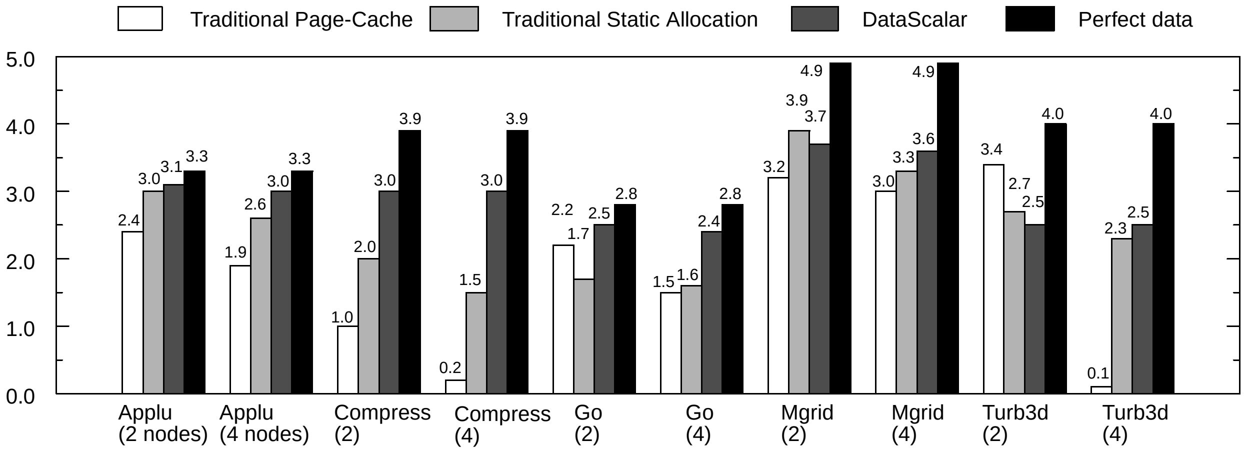


FIGURE 8. DataScalar vs. single processor with static on-chip memory allocation

The DataScalar system deals with a finer-grain distribution of memory better than does either of the traditional systems; the drops in DataScalar performance when going from two-processor to four-processor systems are less than 0.05 IPC (the comparable drops in performance on the traditional system with static memory allocation range from 0.1 to 0.6 IPC while performance drops on the traditional system with a page cache can reach 2.6 IPC). In only two cases (`mgrid` and `turb3d` with two nodes) does the DataScalar system perform worse than the traditional systems. 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).

3.4 Sensitivity analysis

We present the results of a sensitivity analysis in Figure 9. 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 8, 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 five of the systems we measured in Figure 8 (perfect data cache, two- and four-processor DataScalar machines, and traditional systems with static memory allocation 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.

4 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 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 node, they are not broadcast: the result is computed locally only and broadcast to the other nodes. If the result is part of the local com-

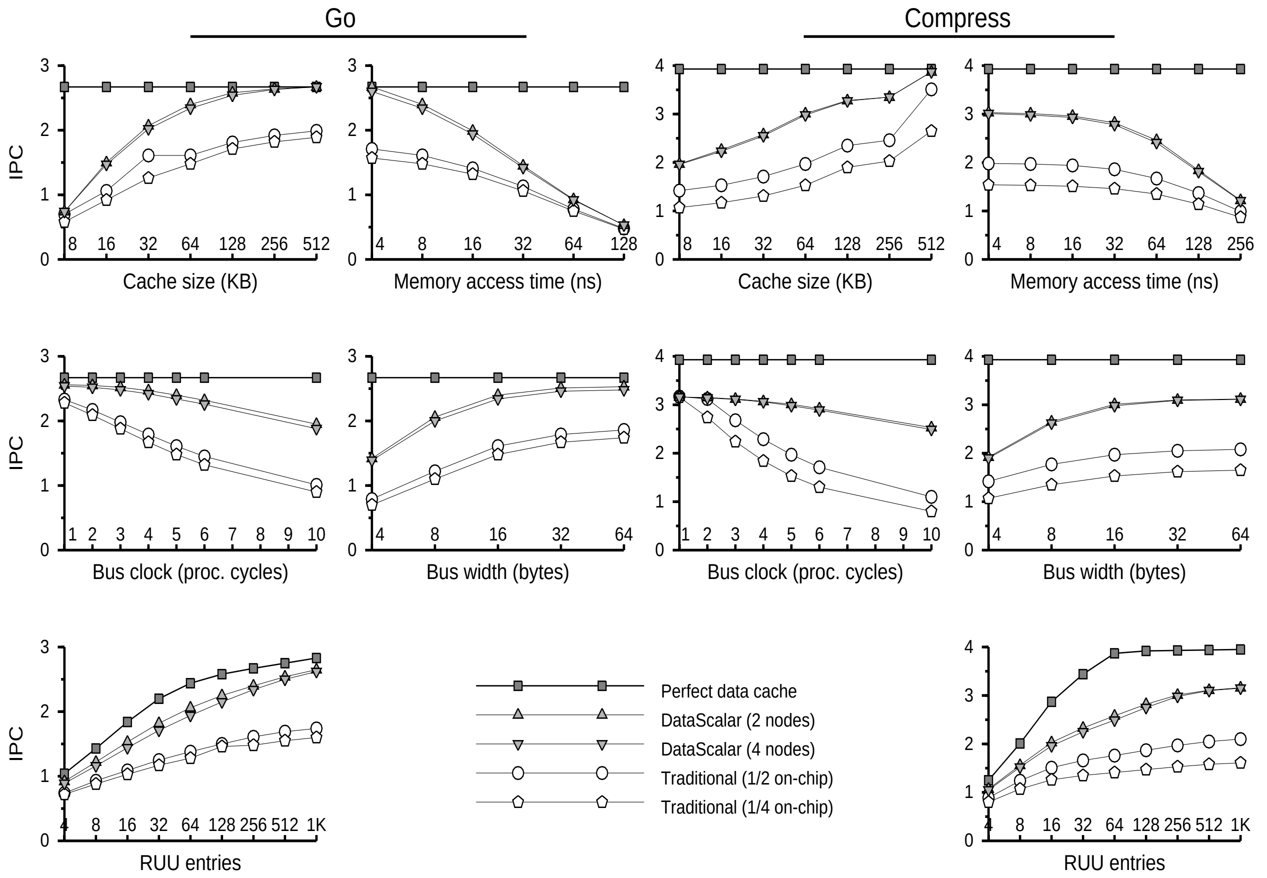
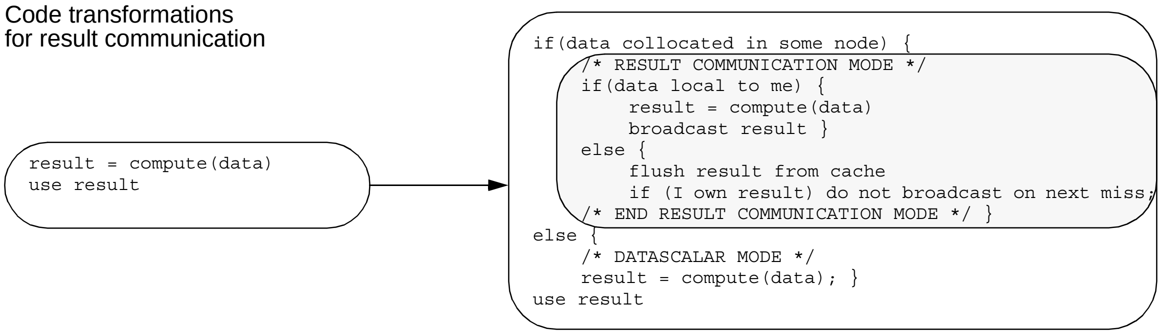


FIGURE 9. Sensitivity analysis for DataScalar and a single node with static on-chip memory allocation

municated data (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; in fact, the other nodes would only compute the result, then throw it away. 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 as communicated data on one node 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 and broadcast it. The software support would be needed to place the entire array on the same node, and to have the other nodes bypass the code that is only to be run on the owning node. 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. Here we present result communication based on run-time locality tests and we illustrate it with simple examples.

Unlike ESP and datathreading, result communication does not come for free with a DataScalar architecture. It requires compiler and run-time support, and could benefit from operating system, programmer, and hardware support as well. To actually implement result communication, the compiler must modify block of code that computes a result and insert two tests (Figure 10). The first test (collocation test) simply determines whether result communication will be used for the computation. This test either succeeds in all nodes or fails in all nodes. It tests whether a group of data are *all* local to *some* node. If the test fails in all nodes, i.e., not all the data are local to any specific node, the computation takes place in the DataScalar mode. Otherwise, if the data are *all* local to some node, the nodes will execute in result communication mode. Each node will then test whether the data are local to it or not (locality test). This test will fail in all but one node which will perform the computation. The nodes that fail the test make sure that they do not have `result` cached. If the owner of `result` fails the locality test it will not broadcast it on the next miss. To implement the two tests (result communication test and locality test) we use a field (*owner* field) in the page table entries to encode the owner node of the page. If the page is statically replicated then a special code that denotes all nodes as owners is used. To test whether multiple data are local to some node we test whether the owner fields of all the relevant pages are the same. To test whether data are local to a specific node we compare the node identifier to the owner fields of the relevant pages.



```
result = compute(data)
use result
```

```
if(data collocated in some node) {
     /* RESULT COMMUNICATION MODE */
     if(data local to me) {
          result = compute(data)
          broadcast result }
     else {
          flush result from cache
          if (I own result) do not broadcast on next miss;
     /* END RESULT COMMUNICATION MODE */ }
else {
     /* DATASCALAR MODE */
     result = compute(data); }
use result
```

FIGURE 10. The compiler transforms code for result communication. Two tests are employed. The first is to determine whether result communication mode will be entered. The second will differentiate among the computing node and the non-computing nodes.

The code in result communication mode must not perturb the state of the correspondent caches, if the main caches are being used to implement dynamic replication. This can be accomplished by marking those accesses in result communication mode, and dropping them at commit-time in the ICUB and DCUB, rather than updating the caches. The code in result communication mode must also force a broadcast on a communicated datum (`result`) that the executing node might not own (although we could fall back to a request/response model in this case). The most important guarantee the compiler must make, however, is that there are no side-effects in the computation that should change the state on all nodes, but only changes it on one node because of the result communication mode.

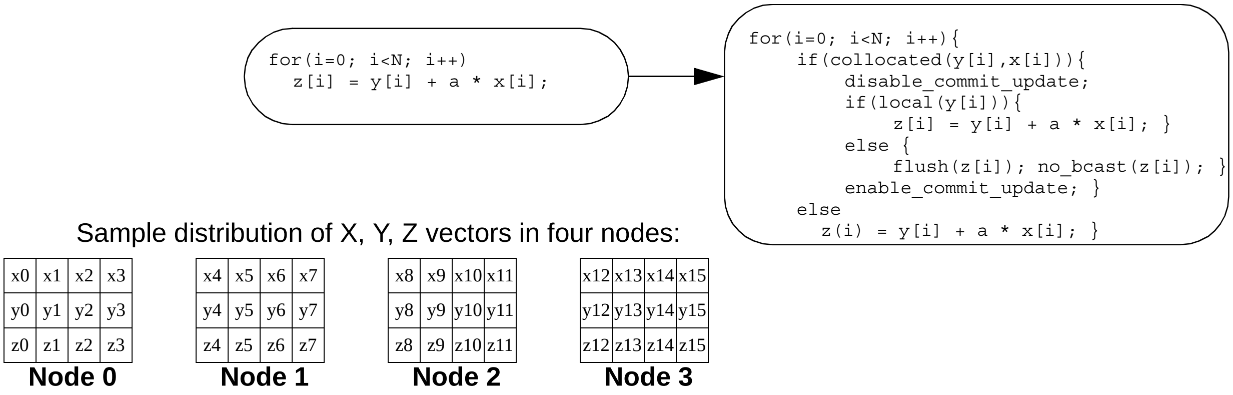
To make effective use of result communication, the run-time system must be able to place certain data items together on the same node (so they can be used in a “local” block of code). One way to accomplish this is to modify the run-time storage allocator by giving it the ownership assignation function, and passing it an address on a request for storage. The storage allocator would make sure to return storage that was allocated on the same node as the address that was passed to it. A more general scheme would be to tag pages with a class identifier, so that the number of nodes (and page assignation) could change dynamically at run-time, so long as all of the pages of a given class came to reside entirely on one node.

With result communication we can parallelize code simply by distributing data appropriately to nodes. The operation of the Distributed Vector Architecture (DIVA) proposed by one of the authors with Sugumar and Schwarzmeier [13] is based on the same principle. We present here an example of a vector computation that is parallelized with result communication in exactly the same fashion as in DIVA.

Lets consider a vector “Single-precision *A* times *X* Plus *Y*” (SAXPY) operation executing in a DataScalar system. The simple loop that encodes the SAXPY on the left of Figure 11 is transformed to the result communication capable code on the right of the same figure. With a proper allocation of the data in memory, each node will go through the full loop but it will execute the computation of only some of the iterations. For example, with the allocation shown in Figure 11 the first node will execute the floating point computation of the first four iterations (0 to 3). Different allocations would lead to different distribution of the floating point computation. In this example the computation is trivial compared to the overhead of executing the full loop and all the tests. In reality, the computation would have to be larger for significant overlap to occur among the nodes. With more computation per iteration, node 0 would be busy executing the first iteration while the rest of the nodes would run down the loop until they found their own part of the computation to perform (e.g., node 1 would run quickly to iteration 4 and start computing from there, node 2 would start at iteration 8, and node 3 at iteration 12). Result communication is thus able to parallelize code according to data placement (we refer to this as *memory parallelism*) which is especially promising for vectorizable/parallelizable code.

5 Conclusions

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 replicates the execution of instructions while it distributes operand accesses across multiple nodes. 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.*,



```
for(i=0; i<N; i++)
  z[i] = y[i] + a * x[i];
```

```
for(i=0; i<N; i++){
    if(collocated(y[i],x[i])){
        disable_commit_update;
        if(local(y[i])){
            z[i] = y[i] + a * x[i]; }
        else {
            flush(z[i]); no_bcast(z[i]); }
        enable_commit_update; }
    else
      z(i) = y[i] + a * x[i]; }
```

Sample distribution of X, Y, Z vectors in four nodes:

x0	x1	x2	x3
y0	y1	y2	y3
z0	z1	z2	z3

x4	x5	x6	x7
y4	y5	y6	y7
z4	z5	z6	z7

x8	x9	x10	x11
y8	y9	y10	y11
z8	z9	z10	z11

x12	x13	x14	x15
y12	y13	y14	y15
z12	z13	z14	z15

FIGURE 11. Parallelization of a loop in result communication mode

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 three 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 three benefit categories—ESP gains, datathreading, and result communication.

Many of our ideas were inspired by the Massive Memory Machine proposal, from which we obtained the concept of ESP [10]. Other research efforts are examining the running of uniprocessor programs much faster by using multiple program counters; the Multiscalar group at Wisconsin [8,26] 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 [19], the PPRAM project at Kyushu University [18], and work at Sun Microsystems [20]. Also, Mitsubishi has developed a multimedia processor prototype integrated with on-chip DRAM [22].

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 higher than either current uniprocessors or parallel processors achieve alone.

6 Acknowledgments

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

7 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, 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.

[3] Doug Burger, Stefanos Kaxiras, and James R. Goodman. DataScalar Architectures. *24th International Symposium on Computer Architecture (ISCA)*, June, 1997.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] Manoj Franklin. *The Multiscalar Architecture*. PhD thesis, University of Wisconsin, Madison, WI, December 1993.

[9] 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.

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

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

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

[13] Stefanos Kaxiras, Rabin Sugumar, James Schwarzmeier. Distributed Vector Architecture: Beyond a Single Vector-IRAM Workshop on Mixing Logic and DRAM: Chips that Compute and Remember, Denver, Colorado, June 1, 1997

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

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

[20] 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.

[21] 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.

[22] 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.

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

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

[25] 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.

[26] 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.

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

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

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