Enhanced Multiprocessor Memory Operations for Fine-Grain Data Sharing and Synchronization*

David M. Koppelman**

Abstract: The growing disparity between instruction issue rates and memory access speed impacts multiprocessors especially hard under certain circumstances. For example, the time to increment a shared counter can include not just the time to get an exclusive copy of the counter's memory, but also the time needed to re-send requests denied because a copy was held by some other processor incrementing the counter. To alleviate the problem a system is described here in which the operations performed by memory controllers are extended so that certain tasks can be completed with less contention, fewer messages, or by avoiding synchronization that would otherwise be necessary. These operations, issued in only a few cycles by a CPU, direct the enhanced memory controller to read, modify, and write a memory location. Tag values can be used to delay completion of an operation until the needed tag value is set. Operations can have multiple steps, and can be sent between memory modules to complete. A completion counter at the issuing CPU can be used to wait until in-progress operations have completed. Demonstrating their usefulness, execution-driven simulation of such systems shows speedup of well over two times on code fragments chosen for their suitability. The radix sorting program from the SPLASH-2 benchmark shows over 20% reduction in execution time.

Index terms: Multiprocessor, latency hiding, memory, active messages, tagged memory.

1 Introduction

Multiprocessors, shared-address-space parallel computers using coherent caches, can efficiently execute a wide variety of code. There are, however, some code sequences take much longer to execute than communication latency might suggest. Consider two examples: In the first, many tasks simultaneously attempt an atomic operation on the same location. Since the tasks (on many systems) must re-try the atomic operation until they get access to the location, reducing the time to complete the operation would speed the operation and reduce the wasteful re-try traffic. In the second, tasks wait in a barrier until all data is ready, even though computation could proceed using data that is ready. The barrier time must be endured because the time and traffic needed to identify ready data would outweigh any savings.

Problems caused in the first example are avoided in some modern systems, such as the Cray Research T3E [26], by having the memory controllers rather than the CPUs execute atomic memory operations. At least one round trip to each accessing CPU is avoided; many are avoided if re-trys would have been necessary. If in the second example the task only applies a few operations to the data, for example, copying to another location, then that processing might also be better performed at the memory. There, processing can be triggered upon data arrival, avoiding synchronization while being able to operate on data as soon as it arrives.

Such a memory processor is described here; it can perform the basic operations needed to implement synchronization and resource allocation already provided in some systems, but also executes instructions that operate on data (*e.g.*, add a constant to a memory location), with some instructions completing only when memory locations are in a specified state. A system that included such a processor would be able to quickly execute fine-grain data sharing code that

* Submitted to *IEEE Transactions on Parallel and Distributed Systems.*

** This work is supported in part by the National Science Foundation under Grant No. MIP-9410435.

would take a conventional processor much longer. This is demonstrated in simulation experiments reported here. Essentially replacing a memory controller, such a processor fits naturally into a multiprocessor system, needing none of the new network or memory bus bandwidth that an added processor would require. By limiting the context that each operation requires, execution overhead is minimized and the memory processor is kept simple.

The enhanced memory controller described here is called an *exo-processor*; instructions intended for exo-processors are called *exo-ops*. *Exo-ops* are designed to be issued by a CPU quickly, for example by writing to an alternate address space. The network interface at the issuing processor assembles an *exo-packet* and sends it to an appropriate memory. An exo-op consists of one or more steps; each step specifies a memory address and an operation; the operation is executed at the address's home memory. By associating tag fields with memory locations, operations such as "write when empty" are easily implemented. A completion counter at the CPU allows code to wait for exo-op completion with almost no overhead.

An exo-processor provides an efficient execution mechanism for many useful functions that could not easily be provided by other means. This is because messages between CPU and memory for simple processing that could easily be performed at the memory are avoided. Unnecessary caching is also avoided. Another reason is the ability to snoop changes to memory locations and quickly react when a location with a pending operation is found.

An exo-processor is a form of smart memory, which has been discussed for some time [30]. As discussed in Section 6 [p. 12] earlier smart memory systems have been either specialized, for example performing image-processing operations, or did not demonstrate a clear advantage over having all processing done by the CPUs [3]. By focusing on operations that are inefficient in multiprocessors, keeping exo-ops small, and by encapsulating all context in the exo-packet and memory location, the exo-processor is made useful and inexpensive enough for practical consideration.

The remainder of the paper is organized as follows. Preliminaries appear in the next section, followed by Section 3 [next page] in which details on exo-processors and exo-ops are presented. In Section 4 [p. 5] cost and performance factors are discussed. Simulation experiments are described in Section 5 [p. 7]. Related work is reviewed in Section 6 [p. 12] and conclusions appear in Section 7 [p. 14].

2 Preliminaries

The following terminology will be used; see [24,29] for details. A *multiprocessor* is a parallel computer that efficiently supports a shared address space for communication between the parts of a parallel program, *tasks*. Such a system consists of *CPUs*, where the programs run, and *memory modules* or *memories* for short, where data is stored. (The term CPU is used instead of processor to avoid confusion with the term *memory processor*.) An *interconnect* provides communication paths between CPUs and memories; a *network interface* connects CPUs and memories to the interconnect. (A CPU is typically paired with a memory module with which it can communicate without using the interconnect.) The network interface prepares and sends *messages* in response to requests by the connected CPU or memory, it also receives messages and dispatches them to the connected CPU or memory [24,29]. The hardware that processes messages received for memories is called a *memory controller* if its simple and a *memory processor* if it is more complex.

For purposes of memory management, the address space is divided into *blocks*, the smallest cacheable unit. Each address is mapped to a memory module called its *home memory* at which it typically is stored. User programs *access* memory (by issuing load and store instructions); systems are designed to implement some *memory model*, which describes the result of such accesses. Caches are provided to reduce access latency; a cached copy of a block is called a *line*. Changes to caches and memory follow a protocol which helps to implement the memory model, typically by insuring that all valid cached copies of a block are identical [24].

3 Exo-Processor

3.1 Hardware

A system using exo-ops consists of a conventional multiprocessor with exo-processors added near memories. In the work reported here addresses operated on by exo-ops are assumed uncached. This may be implemented by making the addresses uncacheable, having exo-ops invalidate cached copies, careful user management, or some other mechanism.

The exo-processor shares access to the memory with the memory controller which processes messages received from the network interface. In fact, the exo-processor might replace the memory controller, handling both normal and exo-op messages. CPUs have a mechanism for fast message assembly and dispatch. Any general-purpose mechanism will do; the description below is for systems that use an alternate address space to interface with a network interface controller that prepares and sends messages. CPUs also have an *in-progress counter;* typically incremented by the CPU and decremented in response to incoming exo-op completion messages. The CPU can wait for the count to reach zero; that is, it can execute an instruction which idles the CPU until the count reaches zero. (Because of their low context switch overhead, multithreaded systems might switch to a new thread. Switching between processes would require either multiple in-progress counters or saving and restoring counter values and special processing when completion messages find their processes no longer running.)

A CPU initiates an exo-operation by executing an instruction identifying the operation (*e.g.*, by writing to an alternate-address-space [AAS] address indicating that the exo-operation is to be started, see below). Operands are specified on that and following instructions. This data, called an *exo-packet*, is transfered to the network interface. At some point in the issue process the in-progress counter may be incremented. Exo-ops have one or more *steps*, typically consisting of a memory access (load or store) and an arithmetic operation.

Exo-operations are envisioned as short, consisting of one to three steps each. The operations (sequence of steps in an issued packet) might be predefined (*e.g.*, hardwired) or could be defined at run time by the parallel program. In the latter case, code implementing the operations would be written to the exo-processors before being issued. (The overhead involved would preclude single-use definitions.)

The exo-packet is sent by the network interface to the memory unit at which the first operand is located. There it is buffered until the first step is executed by the memory's exo-processor. If the exo-packet contains additional steps it is sent to the memory at which its next operand resides (if the next operand is not at the same memory) and the process is repeated. After the last step a message to decrement the in-progress counter may be sent to the issuing CPU.

The utility of exo-ops is greatly increased by the use of tagged memory, in which several bits of state are associated with each block. Exo-operations might wait in an exo-processor until a tag on a needed block takes on a particular value, indicating their presence in the block's directory. This could be implemented by writing a field in a block's *state record* with the storage index (within the exo-processor) of an exo-packet that did not find the expected tag; the expected tag might also be written. Any access that changes a block's tag would check these fields so any now enabled exo-packets found could be resumed. Further implementation details are omitted.

As illustrated below, such tags can be used to implement very efficient synchronization and atomic operations. When tag bits are part of the block (as opposed to extra bits added to the storage defined for an address) implementing tagged operations adds little to the cost of a system that already includes an exo-processor.

3.2 Operation Issue and Execution

The issue and execution of exo-operations by the CPU will be illustrated using two examples in which data to be operated on is widely scattered. Assume that the time to rearrange the data for efficient execution would be comparable to the time needed to operate on the data once arranged. Using exo-ops either rearrangement time or inefficient access is avoided.

Consider the following assembler pseudocode:

Code Fragment 1

```
 LOOP:           ! Label for start of loop.
   add r1,1,r1   ! r1 = r1 + 1, compute address of b[i].
   ld  [r1],r6   ! Load b[i] and place in r6.
   add r6,r2,r6  ! r6 = c + b[i], add a constant, r2, to the first operand.
   add r3,r5,r3  ! i = i + f, the destination address. (f is large.)
   st  r6,[r3]   ! Store the result at address of b[f*i], r3.
   sub r1,r4,r0  ! r0 = r1 - r4, subtract to set branch condition.
   b nz, LOOP    ! Continue if branch condition not zero.
```

to be executed on a conventional multiprocessor. The code computes `a[f*i]=b[i]+c` over a range of `i`, where `f` and `c` are constants. With `f` large the write may miss every iteration; if `a[f*i]` is never accessed again by that CPU a useful line may be evicted for no reason and the CPU that does access the data will have to perform an invalidation.

In an exo system an exo-packet containing addresses of the array elements and the constant `c` is sent first to the memory holding `b[i]`; the exo-processor reads the element, adds `c`, writes the sum to the exo-packet, and sends it to the memory holding `a[f*i]`. There, the exo-processor writes the result and sends an acknowledgment which decrements the in-progress counter at the originating CPU.

The CPU issuing the operation does not have to wait for it to complete (although it could have by waiting for the in-progress counter to return to zero). Since the exo-op bypassed the cache useful lines were not replaced and a second CPU reading the result would not have to wait for an invalidation to complete. If tags were used, a CPU could read the result as soon as it became available without resource-consuming spinning.

The assembler pseudocode below implements the loop using an exo-op:

Code Fragment 2

```
LOOP:            ! Label for start of loop.
    add r1,1,r1  ! r1 = r1 + 1, compute the address of the first operand.
    add r3,r5,r3 ! r3 = r3 + r5, compute destination address. (r5 is large.)
    exo A4,0, r2 ! Initiate exo issue, transfer constant operand, in r2.
    exo A4,1, r1 ! Transfer address of first operand.
    exo A4,2, r3 ! Transfer address of destination.
    sub r1,r4,r0 ! r0 = r1 - r4, subtract to set branch condition.
    b nz, LOOP   ! Continue if branch condition not zero.
```

Here, `exo A4,`**`s`**`,` **`reg`** specifies exo-operation `A4`, step **`s`** using data from register **`reg`**, where `A4` indicates an add with one immediate and one indirect operand. (The assembler would translate these synthetic instructions into writes to the alternate address space used by the exo-processor or network interface, using `A4` and **`s`** to form an address and the contents of **`reg`** as the data.) In the first step, `exo A4,0, r2`, the contents of `r2` is transfered, in the second and third steps the contents of `r1` and `r3` are transferred.

Once the three parameters of `A4` are issued the CPU continues with the subtraction while the exo-processor processes the exo-operation or the network interface prepares and sends the exo-packet. An exo-packet consists of an identifier for the operation, and if necessary, something acting as a program counter, and data.

The operations to be performed by the exo-processors themselves will also be illustrated using assembler code. (This code is not to be taken as the actual instructions or the number of actual instructions.) Regardless of implementation method, the actual instructions would be placed at the exo-processor in advance. The exo-packet constructed for the example above includes a field specifying the operation and three fields for the data: operand 1, address of operand 2, and the destination address; these will be respectively referred to as `x0`, `x1`, `x2`.

Code Fragment 3

```
x_ld  [x1],x1       ! Read [x1] and place in x1 (replacing the address).
x_add x0,x1,x0      ! Add (replacing constant).
x_st  x0,[x2]       ! Store sum.
```

The first instruction replaces field 1 with the data at the address specified by field 1. The second instruction performs the add and the third writes the result. The load and add might be executed at one memory and the store might be executed at another memory.

Instructions for tagged operations will be specified using a *precondition* and *postsetting* in parenthesis as follows: `x_ld [x1],x1 (pre->post)`. The load operation will wait until the tag at `x1` is set to `pre`; reading the data will set the tag to `post`. Operations with $\star$ in the precondition position are unconditional; a $\star$ in the postsetting position indicates that the tag is not to change.

Tags can be used to implement many operations, permutation will be illustrated. Suppose certain elements of an N-element array are to be permuted. The array can be permuted in place (without using additional address space, but possibly using a large amount of exo-processor storage) by issuing the following exo-op:

Code Fragment 4

```
x_ld  [x0],x0: (0->1)   ! Read "from" location, set tag.
x_st  x0,[x1]: (1->0)   ! Write at "to" location after "old" element read.
```

for the elements being permuted. Note that another permutation on the same elements would be issued with opposite preconditions and postsettings. To perform the permutation conventionally the permutation would have to be factored into transposes, the entire array would have to be copied, or the permuted portion would have to be copied back. See Section 5.2 [p. 8] for permutation performance figures.

4 Feasibility

To argue feasibility, the cost of the exo-processor itself will be bounded and its storage requirements and speed estimated. To show that exo-operations do not overload processor/cache and network bandwidth, the resources consumed by the issue of an exo-operation will be estimated (complementing the simulations described in Section 5 [p. 7]).

4.1 Implementation Alternatives and Cost

The exo-processor may be implemented using a general-purpose, dedicated processor or as something like an interrupt handler on the CPU. Also possible, a special-purpose, dedicated processor capable of reacting quickly to completion of memory operations and packet arrivals could be used. The addition of a dedicated processor begs the question: why not build a conventional system using twice as many CPUs? The question implies that system cost is primarily the cost of the CPUs themselves; but for each CPU there is a cache, memory module, and interconnect capacity. Therefore a system using twice as many CPUs would add much more cost than adding an exo-processor at each memory unit.

In a simple implementation the CPU performs the functions of the exo-processor; tag and other information is part of normal storage. Additional hardware for such a system would be

minimal, for example a high-speed message send and receive mechanism (as envisioned for various active-message systems). Since no new processors, storage, or datapaths would be needed, the increased hardware cost is insignificant. As shown by the simulations, performance improvement is attainable even at low speeds.

The amount of storage needed per exo-processor will be estimated by considering the maximum number of in-progress exo-operations. In the simulations performed, the number of in-progress operations was limited to 25; the storage for 100 in-progress operations will be computed here. A five-operand packet of 64-bit operands and containing a 64-bit header is 48 bytes; add 16 bytes for storage needed during execution. Then 100 such packets consume 6,400 bytes, a small amount even for the high-speed storage needed.

4.2 Exo-Processor Speed

As will be shown in Section 5 [next page], performance is sensitive to exo-op execution rate. The following shows how adequate rates might be achieved using a general-purpose processor and high rates with custom hardware.

As envisioned, packets are executed by the exo-processor while the network interface copies others in to and out of its memory, saving time. To save memory allocation time, packet size is bounded and this amount of space is used for each packet, regardless of its actual size. Storage is allocated in circular fashion; an arriving exo-packet may find its storage occupied, since packets are removed at arbitrary times this can happen when free space is available. If so, a trap routine would compress storage or wait until the space is free. The exo-processor can issue transactions to the shared memory itself in a single cycle without blocking. The exo-processor can quickly switch from processing one exo-packet to another (*e.g.*, issuing a memory transaction for one then performing the operation for another) by, for example, avoiding a context switch.

Assembler pseudocode written to estimate the performance of a general-purpose processor implementing exo-processor functionality uses about twenty instructions per exo-op. For example, a step that reads a location, adds a constant, and sends the sum to another exo-processor would use 18 instructions: 8 to find the packet and start the memory read; after performing some other function, 3 to jump back to the code implementing the step; and 7 more to read operands, perform the add, write the result to the packet, and to signal the network interface to send the exo-packet. Tagged operations take a few more instructions, for example, 41 instructions (about 20 each) for these two exo-ops: a tagged write needing an empty location preceding an emptying read to the location.

Faster performance is obtained with a custom processor. For example, if a queue were used for pointers to ready exo-packets (newly arrived or memory transaction complete), 7 instructions per packet could be saved. In fact, specialized hardware using dedicated datapaths and multiported storage for exo-packets could possibly execute at the rate of one step per 3 cycles (not instructions) or more.

4.3 Memory and Bandwidth Consumption

The issuing of exo-ops consumes issue slots and "pin" bandwidth at the CPU and execution consumes communication bandwidth throughout the system. As shown in the analysis below, issuing exo-operations can use about the same number of issue slots and amount of pin bandwidth as executing conventional instructions. Executing exo-operations may take more or less network bandwidth than conventional instructions; in the conventional implementation of Code Fragment 2 exo-operations would require less network bandwidth when line sizes are long.

The number of instructions to code the addition example is the same for the conventional and exo-op implementations of Code Fragment 2. The number of cycles to execute each iteration is limited by restrictions imposed by the memory model, usable issue slots, and functional-unit

availability. Assuming that load/store functional units (which can access the alternate address space) are the limiting resource, one additional cycle is needed to issue the exo-operation—assuming no cache misses. Other uses of exo-operations take fewer cycles to issue than the conventional code they replace, for example Code Fragment 2 with all operands indirect. Overall, issue time of exo-operations is comparable to the operations they replace, and so does limit CPU performance.

Assuming 16-byte cache lines, $s_{\mathrm{L}} = 16$, a 4-byte header, and 4-byte addresses, the conventional implementation would generate 40 bytes of traffic per iteration (each miss results in an 8-byte request and a $(8 + s_{\mathrm{L}}/4)$-word response, each iteration generates $1 + 4/s_{\mathrm{L}}$ misses) and the exo-op implementation would generate 32 bytes (the exo-packet to read the first operand is 16 bytes, the packet to write the result is 12 bytes, and the acknowledgment is 4 bytes).

In some cases exo-operations may generate more traffic; when the cache accesses the exo-ops replace hit, much more, and so exo-operations would not be appropriate. In other cases, for example, when a CPU (or its cache controller) would issue re-trys, exo-operations would generate much less traffic. Based on issue slots and traffic, exo-operations used properly consume at worst marginally more resources than conventional approaches, and as shown below, not enough to undermine performance.

5 Simulation Experiments

5.1 Methodology

Simulation experiments were performed to determine the performance of code fragments and a benchmark with and without exo-ops and to determine which bottlenecks limit performance. Simulations were performed using a modified version of Proteus [4], an execution-driven parallel computer simulator.

Under Proteus *user* programs to run on the simulated system are written in C using special functions for parallel execution such as task creation and shared-memory allocation. Programs are compiled into assembly code using a host system compiler, in the work reported here, gcc 2.7.0. In preparation for simulation the assembler code is augmented, that is, additional assembler code is inserted. The augmented code is assembled and linked with Proteus' own code. The simulator along with the augmented user program execute as a multithreaded program, with the threads managed by the simulator. There is at least one thread for each task, plus one thread for the simulator itself. Some inserted code keeps track of simulated time and initiates a context switch to the simulator when the thread's *quantum* is complete, keeping the simulated time of all threads within a narrow range. Many parallel system functions also initiate thread switches.

Simulated program instructions that do not access memory currently take one cycle of simulated time to execute, see Section 5.1.1 [next page] for timing details on other instructions. A cached shared memory system is fully simulated; the protocol used is similar to the one described in [5]; sequential consistency is maintained, except of course for locations accessed by exo-ops. The interconnection network is simulated at the packet-transfer level.

The version of Proteus used for this study, Proteus L3.8, is a modified version (in addition to the exo-op modifications) of Proteus Version 3.1. Relevant modifications are in the handling of shared memory and compiled code. In the modified version, access to shared memory is specified in C source code in the same way as access to ordinary memory; at the assembly language level load and store instructions that might access shared memory are replaced by code that tests for and if appropriate simulates shared memory access. (In version 3.1 shared-memory accesses are replaced with function calls before being compiled.) The programs to run on the simulated machine can be compiled with optimization turned on. When a line is to be evicted from the cache, an invalid or empty line is chosen, if no such lines exist the line is chosen randomly. For details on these and other modifications see [20].

Table 1: Base Configuration Parameters

Simulation Parameter	Value
System Size	16 CPUs
Network Topology	4×4 mesh
Cache Size	2^{12} sets
Cache Associativity	8
Cache Line Size	16 bytes
Cache Capacity	524,288 bytes
Cache Hit Latency	3 cycles
Mem. Mod. Cap.	2^{19} bytes
Address Space Size	32 bits
Directory Size	full map (16)
Memory Latency	9 cycles

Simulation Parameter	Value
Protocol Message Size	8 bytes (plus data)
Network Interface Width	3 bytes
Network Link Width	3 bytes
Wire + Switch Delay	4 cycles
Exo-Packet Size	8 bytes (plus data)
Exo-Op Issue Latency	$1 + 1$ cycle/operand
Exo-Packet Constr. Time	5 cycles.
Exo-Op Step Exec. Latency	20 cycles
Exo-Op Step Exec. Rate	1 step/10 cycles.
Processor Stall Thresh.	25 in-progress
Processor Resume Thres.	24 in-progress

5.1.1 *System Configurations*

The simulated systems are described in terms of differences with a *base* configuration having 16 CPUs interconnected by a two-dimensional mesh network. Link widths are 3 bytes, chosen so that processor/network bandwidth in bytes per executed instruction approximately match the Sun Ultra Enterprise 4000 (assuming the 4-way Ultra Sparc sustains an issue rate of 3). There are 16 2^{19}-byte memory units using full-map directories; address space is organized into 16-byte blocks. At the CPUs, 2^{19}-byte, eight-way, set-associative caches are used. A list of simulation parameters appears in the table above.

Exo-ops with p operands take $p+1$ cycles to issue (*e.g.*, the three `exo` instructions in Code Fragment 2 would together take four cycles) another five cycles to construct, and twenty cycles to execute each step. An exo-processor can start executing steps every ten cycles. The additional time needed for cache and memory transactions and network transit time is fully simulated and part of operation timing. To limit hot spots, processors with 25 in-progress exo-ops stall; they resume at 24.

5.2 Illustrative Code Fragments

Five code fragments were written to illustrate exo-operations (not whole-program performance) under favorable and unfavorable conditions: histogram, nested loops, vector sum, and two array permutations. All were compiled with optimization on. The fragments and their performance on the base system are discussed below; their performance on other configurations are discussed in Section 5.4 [p. 11].

5.2.1 *Histogram*

The histogram fragment computes a global histogram of data partitioned among CPUs. Processors increment a bin corresponding to each data element. The code was implemented three ways: using an atomic fetch-and-add instruction that executes at the CPU, spin locks, and exo-operations. (In an alternative method a local histogram would be computed and then consolidated into the global histogram; this would take too much time when samples are sparsely distributed among bins.) In the histogram fragment conventional techniques are particularly inefficient and so the exo-op code is much faster.

The execution of fetch-and-add gets a writable copy of the block into the cache and then performs the add; execution time is the same as an ordinary write. The spin lock uses a test-and-test-&-set algorithm. A 1600-bin histogram was computed, elements were randomly distributed

over bins with a uniform distribution. Bins were protected using 64 spin locks, each covering 25 bins.

Execution time for the fetch-and-add and spin-lock implementations were 101 and 367 cycles per iteration, respectively. The exo-operation implementation required only 17.5 cycles per iteration, almost six times faster than fetch-and-add and much faster than spin-locks. Relative benefit improves with increasing line size and remains about the same with low-bandwidth networks.

5.2.2 *Nested Loops*

The nested-loops fragment computes a new array using two values from the old (new in the previous iteration) array; in the conventional implementation

Code Fragment 5

```
for( outer=0; outer<outer_end; outer++){
  if( outer & 0x1 ){nl_x=nl_A; nl_y=nl_B; } else {nl_x=nl_B; nl_y=nl_A; }
  barrier();
  for( inner=start; inner<stop; inner++)
     nl_x[inner] =  nl_y[inner] + nl_y[ nl_p[inner] ];}}
```

where `nl_A` and `nl_B` are the base of two arrays, `nl_p` is the base of a mapping of the indices, and `start` and `stop` specify the part to be performed at the CPU. In the conventional implementation the indirect read (using index `nl_p`) will frequently miss after the first iteration and the write will generate invalidations to other CPUs. Further, the overhead of the barrier is significant when `stop-start` is small and regardless, the barrier precludes overlapping of outer iterations.

The exo-op implementation uses a single exo-operation for the inner loop body with tags and three parameters, `&nl_x[inner]`, `&nl_y[inner]`, and `&nl_y[ nl_p[inner] ]`, in which `&base[index]` indicates the address of element `base[index]`. Reads are performed when a tag is at a proper value, *e.g.*, full; writes set the tag. In the exo-op implementation invalidations associated with the write are avoided, barrier overhead is avoided, and outer loop iterations are overlapped.

On the base system, with 16 iterations per inner loop per CPU, the conventional implementation takes 159 instructions per iteration; the exo-op implementation takes 64.6, almost 2.5 times faster. This fragment puts a substantial load on the memory system because the exo-packets are large and each exo-operation has three steps. The limit on performance here is the rate at which exo packets can be transfered across links and the memory-access rate. Each exo-packet, 20 bytes, takes 7 cycles to transfer; the acknowledgment, 8 bytes, takes 3 cycles. For each exo-operation three exo-packets and one acknowledgment are received (a network interface is shared by a processor/cache and memory module), averaging 24 cycles per iteration per network interface. For each operation total memory time is 27 cycles and exo-processor time is 30 cycles. Because of underutilization due to data location irregularities and stalling of CPUs with 25 in-progress operations each iteration takes 64.6, rather than the ideal 30 cycles per iteration.

5.2.3 *Vector Sum*

The vector operation fragment computes `a[i] = a[i] + b[i] + c`, with each CPU getting a range of `i` values. This is an operation that shared-memory machines do well and vector machines such as the Crays do best. In the former a miss to the first element on a cache line brings in subsequent elements; in the latter the programmer or compiler, taking advantage of the simple access pattern, would bring data to the processors in advance, using a high bandwidth interconnect that could keep up with the processors. Exo-operations, in contrast, only avoid the first miss. The fragment was implemented three ways: using conventional approaches that miss and hit the cache and using exo-ops.

The conventional approaches that miss and hit took 47.7 and 27.2 cycles per iteration, respectively; exo-operations took 46.0 cycles per iteration, respectively. The exo-op approach is only slightly faster than the conventional code that misses, however for more aggressive implementations the exo-ops margin is larger. For example, when exo-op execution latency and rate are 15 cycles and 1 per 5 cycles respectively, exo-ops took 35.1 cycles per iteration, a better speedup. With slow network links the vector exo-operations with base timing are over twice as fast.

When accesses hit, the conventional approach is faster, as one would expect. With misses, exo-ops are 1.04 times as fast, an insignificant improvement. As before, the limit on performance is transfer and memory access latency; unlike the nested loops, spatial locality reduces the number of misses the conventional code suffers but does nothing for exo-ops. For these reasons in low-bandwidth networks exo-operations take 1.25 times longer and also perform worse at line sizes above 16 bytes and for slow exo-op implementations.

5.2.4 *Permutation*

Two permutation code fragments are simulated. The *in-place* fragment permutes some array elements (leaving the others moved); the *copy* permutation fragment fills a new array with the rearranged elements from the original. In both fragments the permutation itself is stored in private memory; the caches are warmed so that the "from" locations are local. One exo-op per transpose is used for the in-place permutation with tags coordinating writes; the conventional code copies moved elements to temporary storage in local memory (100% hit rate, 1 cycle hit latency), executes a barrier, then copies them back. The copy permutation is implemented with loads and stores; an inverse permutation is used so that writes to local elements do not exhibit false sharing, speeding the conventional code. In all cases the system is initialized so that source and destination (if any) array elements are exclusively cached at the CPU near their home memory. Arrays had 4096 integer elements; the in-place permutation was performed on 800 elements, the maximum that could be performed without risking deadlock (given the base configuration's stall threshold of 25).

Using the conventional approach, the in-place permutation took 205 cycles per iteration; using exo-operations, 56.5, 3.6 times faster. The copy permutation took 72.5 and 31.9 cycles, respectively; over twice as fast. By avoiding false sharing and coordinating writes, exo-ops perform the in-place permutation well. Exo-ops also improve the copy permutation.

5.3 Benchmark

Simulations were performed using a modified version of Radix, a kernel from the SPLASH-2 benchmark suite [32] that implements a radix sort. Suggestions on placement of tasks and allocation of shared memory appearing in the benchmark were followed so that the radix was well tuned, with or without exo-ops. Radix was compiled with optimization, and run using the default problem size specified in the code: 262,144 elements chosen over range $[0, 524288]$ sorted using radix 1024.

Radix uses exo-scalar operations to permute the keys which it is sorting and to compute prefix sums. The prefix sums are computed using exo-ops implementing a linear chain of additions, each addition issued by a different CPU. The prefix sum is computed using about one message per element whereas the conventional code uses two (a read and write). The Radix program was also modified to improve the performance of the prefix sum using conventional operations, the modified program runs much faster than the unmodified version on the configurations tested. The conventional implementation takes 3.38 million cycles; using exo-ops, 2.61 million cycles, a reduction of over 22%. The conventional code spends about 40% of its time ranking and 60% permuting. Exo-ops reduce the permute time by about a third, improvement is limited by read misses to the

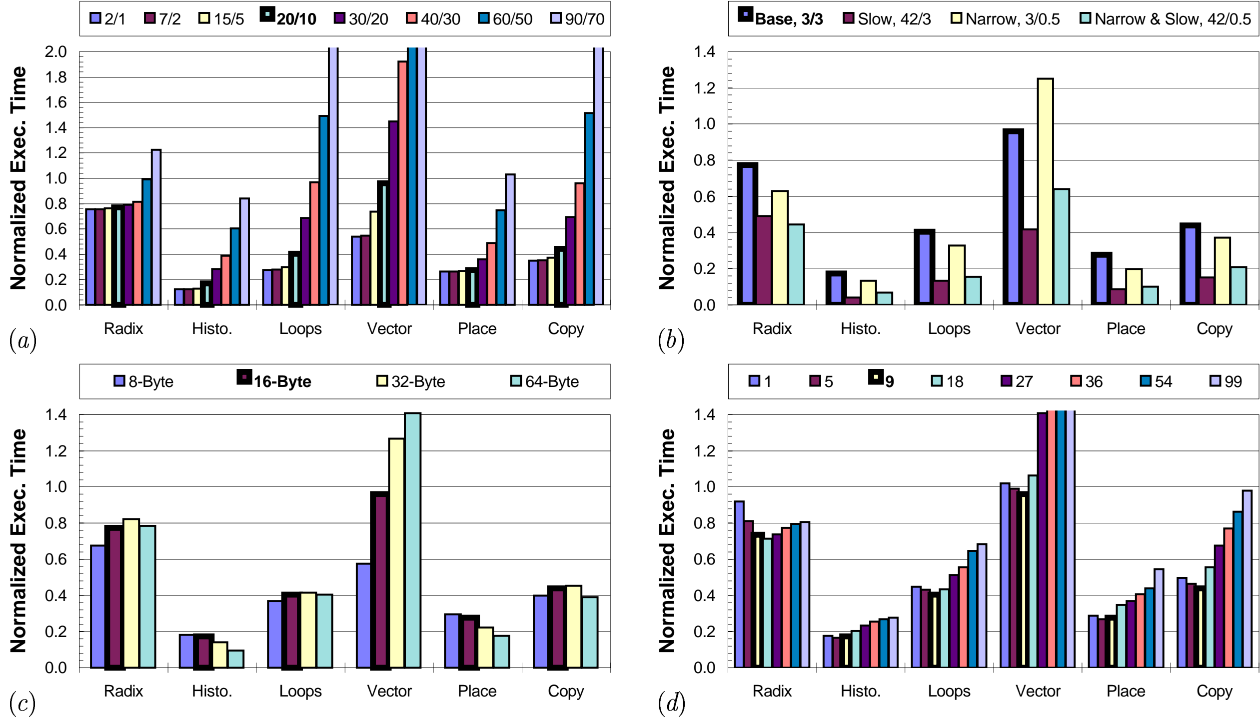

Figure 1. Execution time of exo-scalar systems normalized to conventional systems (*e.g.*, 0.5 indicates half the execution time of a conventional system). Parameters varied are (*a*) implementation latency/issue period (cycles), (*b*) network link delay (cycles) /widths (bytes), (*c*) line sizes, and (*d*) memory latencies (cycles). Bold indicates base configuration.

element being moved. Prefix time is a small fraction of execution time, so its reduction had little impact.

5.4 Configuration Effects

Performance improvement of the benchmark and fragments is plotted in Figure 1(*a*) for a variety of exo-op execution speeds. Key x/y indicates x-cycle latency and y-cycle period (1/rate); base speed is in boldface. (Latency is the time from start to finish of an exo-op step; period is the minimum time from starting one exo-op step to starting another.)

A program's sensitivity to exo-processor speed is determined by the rate at which the program issues exo-ops in comparison to exo-processor execution rate. Radix's insensitivity, as can be seen in Figure 1(*a*), is due to the relatively long time used to compute element destination and the significant amount of time spent executing code that does not use exo-ops. Histogram's insensitivity is due to fast execution of the single-step exo-op which contrasts with vector's three-step exo-op.

Where exo-operations generate traffic, bandwidth limitations could undermine their usefulness; this was not observed in the simulations where a system using 4-bit links (called "narrow" in Figure 1 (*b*)) was compared to the base system (which uses 16-bit links). Relative performance on a low-bandwidth system is improved for all but vector, which generates more traffic than the code it replaces. Because exo-ops are non-blocking, relative performance on all benchmarks is much better on high-latency networks with high or low bandwidth.

The effect of line size is plotted in (*c*). Line size primarily affects the conventional programs, longer lines improving performance where there is spatial locality, worsening performance otherwise. Radix suffers both effects: the conventional code is fastest with 32-byte lines, by avoiding false sharing the exo implementation is fastest at the largest line size. False sharing also is

present in the histogram and in-place permutation codes. The conventional copy permutation and loop code suffer from using a small part of the lines they read.

Memory speed effects are shown in (*d*). Exo-ops perform best at middle speeds; with faster memory there is less latency to hide, slower memory is a bottleneck for both implementations. Not shown is system- and cache-size effects; relative improvement is greater on the larger hypercube systems tested, where link width and per-processor cache size are held constant, cache size strongly effected the execution time for Radix on both conventional and exo systems, relative benefit was about the same. Small caches improved the performance of the exo-op implementation of some fragments (since fewer lines needed to be invalidated) while reducing the performance of the conventional implementations.

6 Related Work

The latency tolerance, parallelism, and synchronization that can be achieved with exo-ops can be realized using a combination of existing techniques, however the combination would be less effective and less flexible.

6.1 Alternate Approaches to Latency Hiding

Where used, exo-operations, being nonblocking, hide miss latency. This can also be achieved using multithreading, out-of-order completion combined with a relaxed consistency model, and prefetching. (To be fair, exo-ops would only hide latency for data which is not needed at the CPU.)

Processors can hide (do something useful during) access latency by performing a context switch on an access miss. When context-switch time is small useful work can be performed during the memory accesses. *Multithreaded processors* provide multiple sets of registers and other processor-state storage so that context switches take little time, in some schemes zero cycles. (See [10] for an early description and [1,2,9,22,27,31] for some recent work.) While latency is hidden, the CPU must be able to hold the state of many threads, adding to the cost. Further, there must be several times more threads in the parallel program, potentially reducing the efficiency of those applications which are not embarrassingly parallel. Most of the added cost in an exo-op system is outside the CPU, which may be a desirable tradeoff.

In a system using out-of-order completion and a relaxed consistency model, instructions following memory accesses that do not immediately complete (*e.g.*, due to a cache miss or some consistency action) would not necessarily stall the CPU [11,13]. Memory writes would cause the least trouble since no instruction would have to wait for these to complete. (Reads to the same location as a pending write could be satisfied by a buffer holding in-progress operations.) Instructions following memory reads would only stall if they read a register to be written by the read.

Fully exploiting such out-of-order completion would require elaborate hardware. A cache miss on a location which is held elsewhere in an exclusive state might take hundreds of cycles to complete. It is possible to construct a CPU which could hold such memory access instructions, and the instructions dependent on it, while executing non-dependent instructions, however the cost may be prohibitive. In contrast, in an exo-op system the processor issues memory reads, arithmetic operations, and writes in a single unit to be processed elsewhere; the number of outstanding exo-operations is the only state information to be maintained.

The difficulties of out-of-order completion can be avoided by *prefetching*, moving data into a cache before needed. Prefetching can be accomplished by having the programmer or compiler insert prefetch instructions for data ahead of its use [11,17,23,25,26], by having hardware regularly issue fetches for memory locations of some fixed stride (the stride and timing set by the programmer or even determined automatically) [6,7,26], or, simplest of all, by using long cache lines. For all of these techniques the memory location must be determined in advance of need, which is not

always possible. In contrast, exo-operations are issued as soon as the operands are known, taking advantage of the CPU's ability to determine operands using conventional code; the operations complete when operands are available, taking advantage of the exo-processor's ability to snoop.

6.2 Atomic Operations using Smart Memory

Exo-ops implement atomic operations (such as adding a quantity to a memory location) very efficiently by performing them at the memory. Some existing and proposed systems achieve such efficiency using *smart memory*. Smart-memory systems, which use memory units having some computational capability, are in some ways similar to exo-scalar systems. The Cray T3E [26], a shipping commercial system, and Cedar [21], an older research system in many ways similar to the T3E, use smart memory to implement atomic operations (such as compare-and-swap, fetch-and-increment, and test-and-add) for synchronization but not for computation. Exo-operations are designed to support computation and synchronization. Smart memory designed to perform computation is discussed in Section 6.5 [this page].

6.3 Synchronization and Computation using Dedicated Networks

Some existing systems, such as the CM-5 [15] and T3D [19] use special networks for barriers and reduction, achieving very low latency on these operations. However barriers and reduction occur too infrequently in most programs for there to be a major impact on performance. Exo-operations can also be used to implement barriers and perform reduction, perhaps not quite as efficiently as a dedicated network, but much faster than implementations using spinning or inter-processor interrupts.

6.4 Data Flow

With tags, exo-operations can be issued when and where the identity (*e.g.*, array index) of operands are determined regardless of whether the operands themselves have been computed. Because of the overhead involved, such operations would only be issued where operand availability (in time or space) could not be assured. Fine-grain data flow and task flow is similar in that operations are triggered by data availability, but is inefficient since that is the only execution mechanism [16,22,28]. Exo-ops are part of procedural code and so the programmer does not have to cast the application's control flow into an unfamiliar data-flow paradigm.

Hybrid or large-grain data flow improves efficiency by executing more of the code as conventional processors would. One such proposed system is *T, which is designed so that tasks can be stopped and started quickly when needed remote data arrives [27]. The machine uses separate memory processors which can hold requests until data is ready, at which time a response is sent which includes a *continuation*; this might be used to restart the stalled task that had issued the request. The issuing of an exo-op does not stall the process and so multiple outstanding operations can be issued by a single thread of execution, resulting in higher utilization (where there would be no other tasks to continue on *T) or greater efficiency (where more work is performed in order to provide multiple tasks per processor on *T). The *T machine is to be a hybrid dataflow system (simulation results were not reported), whereas the scheme described here is much closer to a conventional processor.

6.5 Computation Using Smart Memory

Smart memory is usually proposed to perform some amount of computation; proposals vary on the granularity of the computation, from simple operations on single-bit operands to implementation of data structures. By the use of tags, operation generality, and in-progress counters, exo-ops can be used for such computation under a wider variety of circumstances and as demonstrated here, in a way that is useful in modern multiprocessors. Many smart-memory schemes are limited, apply only to specialized systems, or are no longer practical.

In early work in this area, described by Stone [30], arithmetic and logical operations would be performed on cached data by a *logic-in-memory* cache. Since the overhead in issuing logic-in-memory operations could easily exceed the time saved by overlapping their execution with CPU activity they are best performed by the CPU itself.

In more recent incarnations simple operations are performed in parallel on large sets of data or complex operations are performed on smaller sets of data. Examples of the former include image-processing operations. Such systems would include many inexpensive memory processors which when used, would outperform the much smaller number of CPUs; they would be inexpensive enough so that low overall utilization would be acceptable. (See for example, [12].) Simple operations allow processors to be economically fabricated on each memory chip, however their simplicity limits their use. An exo-processor is more complex, but far more flexible, and only one is needed per memory unit (or enough to keep up with a CPU).

In other smart memory schemes the memory would implement data structures, rather than a simple sequence of storage locations [3]. A memory access instruction might request that an element of a linked list be returned, for example. Such ideas have yet to be fully worked out, in particular, what operations should the smart memory support and how would code executing on a smart memory processor outperform code running on a regular processor. Exo-processors are designed for simple operations using little context; a memory processor implementing an elaborate data structure would require a lot of context (registers, stack, etc.) and so would have the cost and task switch overhead of a processor.

6.6 Active Messages

Active messages might be used to implement something similar to exo-ops. Active messages are messages which can be assembled, sent, and processed at their destinations with very low overhead, possibly by directly accessing processor registers. The time needed to process an active message at its destination is kept to a minimum by placing the address of an interrupt handler, or even an opcode, within the message. See [8] for active message implementation for the J-machine [14] for FLASH, and [18] for EM-X. While the issuing of an exo-op is almost identical to the sending of an active message, the interrupt of the CPU at an active message's destination introduces latency and CPU overhead that exo-operations avoid. If a context switch is needed then many cycles are lost—both by the active message and the interrupted task. Exo-operations, in contrast, can be handled by a simple execution unit which includes no more context than is present in the exo-packet and the memory location read, eliminating the need for state saving.

7 Conclusions

A multiprocessor system using an enhanced memory controller, called an exo-processor, as been described and evaluated. Execution of exo-ops by the exo-processor takes advantage of its location by, for example, avoiding memory-processor-memory messages needed only to complete simple operations and by snooping on other accesses to the memory location. Other features allow processors to issue these operations with only a few instructions, and when necessary wait for their completion by stalling until an in-progress counter reaches zero. With this tight integration, exo-ops can be used for fine-grain data sharing that would be impractical on conventional multiprocessors.

Execution driven simulation has shown substantial speedup on code fragments suited for exo-op use; any program which spends most of its time executing similar code would show this impressive speedup. More realistically, a program might spend only part of its time executing such code, as does Radix, in such cases speedup is not as high, but still good. The feasibility of exo-ops depends upon the cost of implementation and the existence of problems which, when appropriately coded, run faster on exo-op systems than on conventional multiprocessors. If the cost and complexity is small, then exo-ops could be included on any multiprocessor; specialized

exo-op systems would be viable if the cost were moderate and the there were enough users with applications that would benefit.

8 References

[1] A. Agarwal, “Performance tradeoffs in multithreaded processors,” *IEEE Trans. on Parallel and Distributed Systems,* vol. 3, no. 5, pp. 525-539, September 1992.

[2] R. Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith, “The Tera computer system,” in *Proceedings of the International Conference on Supercomputing,* June 1990, pp. 1–6.

[3] A. Asthana, H. V. Jagadish, J. A. Chandross, D. Lin, and S. C. Knauer, “An intelligent memory system,” *ACM Computer Architecture News,* vol. 16, no. 4, pp. 12-20, September 1988.

[4] E.A. Brewer, Chrysanthos N. Dellarocas, Adrian Colbrook, and William E. Weihl, “Proteus: a high-performance parallel-architecture simulator,” in *Proceedings of the ACM SIGMETRICS conference,* May 1992.

[5] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal, “Directory based cache coherence in large–scale multiprocessors,” *IEEE Computer,* vol. 23, no. 6, pp. 49–59, June 1990.

[6] Tien-Fu Chen and Jean-Loup Baer, “Effective hardware-based data prefetching for high-performance processors,” *IEEE Transactions on Computers,* vol. 44, no. 5, pp. 609-623, May 1995.

[7] Fredrik Dahlgren, Michel Dubois, and Per Stenström, “Sequential hardware prefetching in shared-memory multiprocessors,” *IEEE Transactions on Parallel and Distributed Systems,* vol. 6, no. 7, pp. 733-746, July 1995.

[8] William J. Dally, J. A. Stuart Fiske, John S. Keen, Richard A. Lethin, Michael D. Noakes, Peter R. Nuth, Roy E. Davidson, and Gregory A. Fyler, “The message-driven processor: a multicomputer processing node with efficient mechanisms,” *IEEE Micro Magazine,* vol. 12, no. 2, pp. 23-39, April 1992.

[9] Jack B. Dennis, Guang R. Gao, and Robert A. Iannucci (Editor), “Multithreaded computer architecture,” Boston: Kluwer Academic Publishers, 1994, Chapter 1, pp. 1–72.

[10] M. Dubois, “A cache–based multiprocessor with high efficiency,” *IEEE Transactions on Computers,* vol. 34, no. 10, pp. 968–972, October 1985.

[11] Kourosh Gharachorloo, Anoop Gupta, and John L. Hennessy, “Two techniques to enhance the performance of memory consistency models,” in *Proceedings of the International Conference on Parallel Processing,* August 1991, vol. I, pp. 355–364.

[12] Maya Gokhale, Bill Holmes, and Ken Iobst, “Processing in memory: the Terasys massively parallel PIM array,” *IEEE Computer,* vol. 28, pp. 23–31, April 1995.

[13] A. Gupta, K. Gharachorloo, T. Mowry, and W. D. Weber, “Comparative evaluation of latency reducing and tolerating techniques,” *ACM Computer Architecture News,* vol. 19, no. 3, pp. 254–263, May 1991.

[14] John Heinlein, Kourosh Gharachorloo, Scott Dresser, and Anoop Gupta, “Integrating of message passing and shared memory in the Stanford FLASH multiprocessor,” in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems,* October 1994, pp. 38–50.

[15] W. Daniel Hillis and Lewis W. Tucker, “The CM-5 connection machine: a scalable supercomputer,” *Communications of the ACM,* vol. 36, no. 11, pp. 30-40, November 1993.

[16] Robert W. Horst, “Task-flow architecture for WSI parallel processing,” *IEEE Computer,* vol. 25, no. 4, pp. 10-18, April 1992.

[17] A. C. Klaiber and Henry M. Levy, “An arch. for software-controlled data prefetching,” in *Proceedings of the International Symposium on Computer Architecture,* May 1991, pp. 43–53.

[18] Yuetsu Kodama, Hirohumi Sakai, Mitsuhisa Sato, Hayato Yamana, Shuichi Sakai, and Yoshinori Yamaguchi, "The EM-X parallel computer: architecture and basic performance," in *Proceedings of the International Symposium on Computer Architecture,* June 1995, pp. 14–23.

[19] R. Kent Koeninger, Mark Furtney, and Martin Walker, "A shared memory MPP from Cray Research," *Digital Technical Journal,* vol. 6, no. 2, pp. 8-21, 1994.

[20] David M. Koppelman, "Version L3.8 Proteus Changes" Department of Electrical and Computer Engineering, Louisiana State University, (simulator documentation), http://www.ee.lsu.edu/koppel/proteus/proteusl_1.html and http://www.ee.lsu.edu/koppel/proteus.

[21] D. Kuck, E. Davidson, D. Lawrie, A. Sameh, C.Q Zhu, A. Veidenbaum, J. Konicek, P. Yew, K. Gallivan, W. Jalby, H. Wijshoff, R. Bramley, U. M. Yang, P. Emrath, D. Padua, R. Eigenmann, J. Hoeflinger, G. Jaxon, Z. Li, T. Murphy, J. Andrews, and S. Turner, "The Cedar system and an initial performance study," in *Proceedings of the International Symposium on Computer Architecture,* May 1993, pp. 213–223.

[22] Ben Lee and A. R. Hurson, "Dataflow architectures and multithreading," *IEEE Computer,* vol. 27, no. 8, pp. 27-39.

[23] R. L. Lee, P.-C. Yew, and D. H. Lawrie, "Data prefetching in shared memory multiprocessors," in *Proceedings of the International Conference on Parallel Processing,* August 1987, pp. 28–31.

[24] David J. Lilja,"Cache coherence in large-scale shared-memory multiprocessors: issues and comparisons," *ACM Computing Surveys,* vol. 25, no. 3, pp. 303–338, September 1993.

[25] Todd C. Mowry, Monica S. Lam, and Anoop Gupta, "Design and evaluation of a compiler algorithm for prefetching," in *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems,* October 1992, pp. 62–73.

[26] Steven L. Scott, "Synchronization and communication in the T3E multiprocessor," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems,* October 1996.

[27] R. S. Nikhil and G. M. Papadopoulos, "*T: a multithreaded massively parallel architecture," in *Proceedings of the International Symposium on Computer Architecture,* May 1992, pp. 156–167.

[28] Vason P. Srini, "An architectural comparison of dataflow systems," *IEEE Computer,* vol. 19, no. 3, pp. 68-88, March 1986.

[29] Per Stenström, "A survey of cache coherence schemes for multiprocessors," *IEEE Computer,* vol. 23, no. 6, pp. 12–24, June 1990.

[30] Harold S. Stone, "A logic-in-memory computer," *IEEE Transactions on Computers,* vol. 19, no. 1, pp. 73–78, January 1970.

[31] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm, "Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor," in *Proceedings of the International Symposium on Computer Architecture,* May 1996, pp. 191–202.

[32] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta, "The SPLASH-2 programs: characterization and methodological considerations," in *Proceedings of the International Symposium on Computer Architecture,* May 1995, pp. 24–36.