A Multiprocessor Memory Processor for Efficient Sharing And Access Coordination

David M. Koppelman

Department of Electrical and Computer Engineering
Louisiana State University, Baton Rouge
koppelffee.lsu.edu

Abstract

The growing disparity between instruction issue rates and memory access speed impacts multiprocessors especially hard under certain circumstances. To alleviate the problem a system is described here in which smart memory chips can execute simple operations 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 memory 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 memories 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 29% reduction in execution time.

1 Introduction

A natural solution to the widening gap between instruction issue rates and memory access speed is adding processing capability to memory chips, where inter-chip delays can be avoided and the large word sizes inherent in memory design is available. The size of, and the role played by, this processing varies. The number of transistors available on a single chip has reached the point at which memory can share a chip with a processor not much less powerful than a conventional single-chip processor [4,25]. These would serve as the main processors in a system. Alternately, the memory processor might be very simple, performing basic operations, but perhaps in parallel on many items fetched at once (perhaps over many chips). (See [12] for a recent example.) Such chips would assist the main processors for specialized tasks, such as image processing.

An intermediate approach is described here, applicable to multiprocessors (cached, single-address-space, shared-memory systems). Memory is coupled with *exo-processors*, which operate on memory locations as directed by *exo-packets*, special messages dispatched by exo-processors and main (non-memory) CPUs. The process starts when the CPU issues an *exo-op*, causing an exo-packet to be constructed and dispatched to an appropriate memory. An exo-op specifies something like a function or complex instruction (depending upon implementation); some data may be placed in the exo-packet itself, other data is stored in the memory (the exo-packet holds the addresses). The CPUs issue exo-ops in non-blocking fashion and may wait for their completion using a completion counter.

Small context (a general term for data accessed during execution) size distinguishes the execution of exo-ops from a thread in conventional code. Whereas a thread requires registers and access to memory, the context needed by an exo-op is contained in the exo-packet itself and the memory it accesses. Small context size enables exo-ops to execute without the need to move much data. Also important for their intended use, it allows a blocked exo-op to be quickly resumed at the appropriate time.

With the co-location of processing and memory, computation can efficiently be triggered by the modification of a memory location. Specifically, exo-ops can block until a particular memory location takes on a particular value. Because of the small context and other design features, an exo-op can resume without taking much time away from other exo-ops. The issuing CPU can use a completion counter to wait until in-progress operations have completed, allowing efficient synchronization. A system using such processors can perform some operations far more efficiently, and can run parallel programs too inefficient for conventional multiprocessors.

Exo-op execution is illustrated in Figure 1 in which the execution of nested loops with (to the left) and without (to the right) exo-ops is plotted. Processor states are shown by shades, light and dark shades are used to

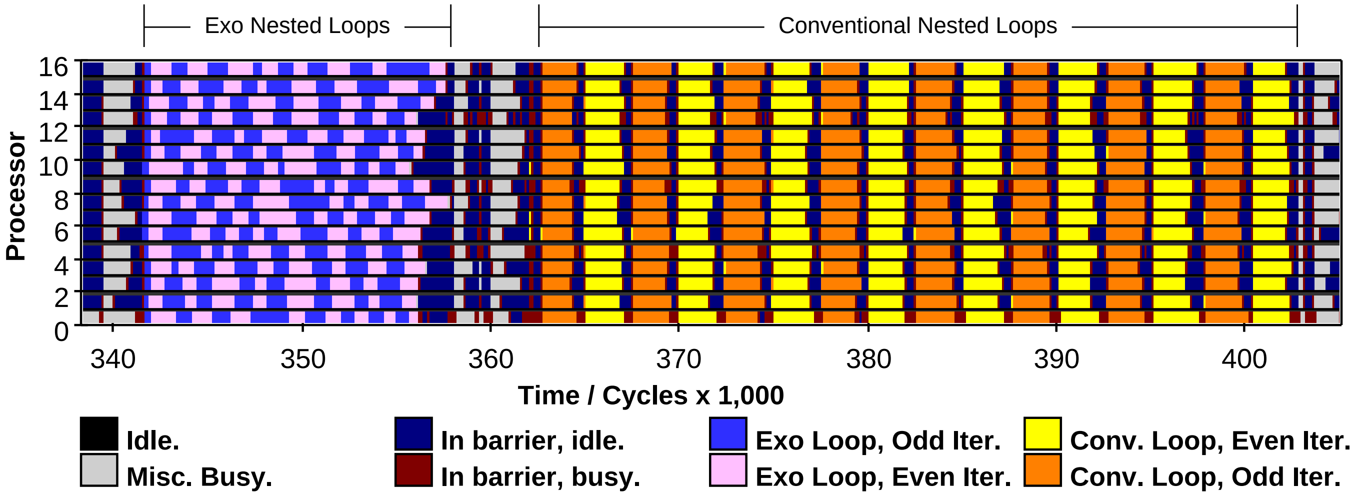

Figure 1. Execution of nested-loops with (left) and without (right) exo-ops. Light shading indicates execution of even iterations of the outer loop, dark shading indicates odd iterations. Execution time without exo-ops is longer because of cache misses and the barrier used in each iteration of the outer loop. With exo-ops, no barrier is necessary and iterations can overlap because tagging is used to ensure the proper operation order.

distinguish even and odd iterations of the outer loop. Data produced by a processor in one iteration of the outer loop is used by other processors in the next iteration. The synchronization is conventionally provided by a barrier, this is shown on the right part of the graph where the synchronization overhead is clearly visible as dark gaps between the iterations. On the left, it can be seen that barrier overhead is not present and that iterations overlap, resulting in faster completion. Overlap is possible because tags are used to distinguish data produced in different iterations. The issuing of exo-ops also avoids cache misses which are frequent in this example. The large non-uniformity in the outer-loop iteration time using exo-ops is due to the processor stalling when the number of outstanding exo-ops reaches a limit.

Exo-ops can be used to avoid the multiple time-consuming communication steps needed to modify data, a benefit when the data will not be needed by the CPU other than for modification. Using the ability of exo-ops to wait for memory to take on specific values, operations can be quickly issued by a CPU before it is known if operands are available. The alternative would be waiting for synchronization and then waiting after cache misses, as the externally produced data is read. To be sure, there are many useful programs where such synchronization and miss overhead is only a small part of execution. But for others the overhead limits speedup.

The remainder of this paper is organized as follows. Preliminaries are presented in the next section, followed by details on exo-processor systems in Section 3. Exo-op effectiveness was evaluated by using execution-driven simulation, these are described in Section 4. Related work is surveyed in Section 5, and conclusions appear in Section 6.

2 Preliminaries

The following terminology will be used; see [20,28] for details. A *multiprocessor* is a parallel computer that efficiently supports a shared address space for communication between the parts of a parallel program. 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 exo processor.) Memory may be divided into independent *banks* such that an access to one bank can start even while another bank is busy. A *network interface* connects CPUs and memories to the interconnection network. The network interface prepares and *dispatches messages* in response to requests by the connected CPU or memory, it also receives messages and routes them to the connected CPU or memory processor [20,28]. 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*. A cached copy of a block is called a *line*.

3 Exo-Processor

3.1 Hardware

A system using exo-ops is derived from a conventional multiprocessor by integrating exo-processors and memories; see Figure 2.

A *memory unit* contains memory, an exo-processor, and a *packet buffer*. A *memory module* consists of several memory units sharing a single network interface, see Figure 3. Arriving exo-packets and other memory requests are sent by the network interface to the appropriate bank

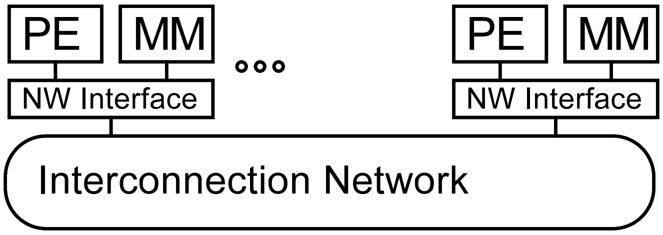

Figure 2. Exo system overview. PEs are processors capable of fast message dispatch but are otherwise ordinary. MMs are memory modules containing banks of memory/exo-processor chips.

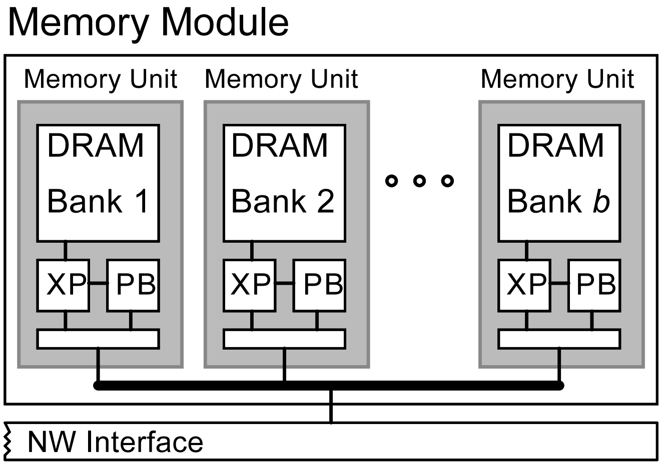

Figure 3. Memory module overview. DRAM, an exo-processor (XP), and a packet buffer (PB) form a memory unit which is implemented on a single chip. A memory module consists of several memory units. In a typical configuration all share a network interface with a processing element.

for processing. In the other direction, memory units dispatch exo-packets and other protocol messages via the network interface. The work performed by one memory unit does not interfere with communication between the network interface and other memory modules, so with enough banks packets can be processed as fast as they can be received.

CPUs have a mechanism for fast message assembly and dispatch, used for issuing exo-ops. 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 dispatch 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 may wait for the counter to reach zero.

A CPU issues 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; they 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.

Upon issue, an exo-op is dispatched to the memory unit at which the first operand is located. There it is buffered (using a simple but inefficient storage allocation mechanism) until the first step can be executed. After execution and if there are additional steps the exo-packet is sent to the memory at which its next operand resides 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 state is associated with addresses. Unlike the tagged memory used in HEP [8], 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 packet buffer) 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.

4 Simulation Experiments

4.1 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 Proteus L3.10 [17] which was derived from Proteus 3.1[3], an execution-driven parallel computer simulator. Code to run on the simulated system is compiled into assembler code (using a host system compiler) and augmented. Augmentation inserts code that, among other things, keeps track of simulated time, performs switches between threads (possibly on different simulated processors), and replaces possible shared memory accesses with calls to simulator routines. Simulated program instructions that do not access memory take one *scaled* cycle of simulated

Table 1: Base Configuration Parameters

Simulation Parameter	Value
System Size	16 CPUs
Network Topology	4 × 4 mesh
Cache Size	2^{11} sets
Cache Associativity	8
Cache Line Size	16 bytes
Cache Capacity	262,144 bytes
Cache Hit Latency	3 cycles
Mem. Mod. Cap.	2^{19} bytes
Address Space Size	32 bits
Directory Size	full map (16)
Banks Per Module	4
Memory Miss Latency	42 cycles
Memory Hit Latency	12 cycles
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	2 + 1 cycle/Word
Exo-Packet Constr. Time	6 cycles
Exo-Op Step Exec. Latency	21 cycles
Exo-Op Step Exec. Rate	1 step/12 cycles
Processor Stall Thresh.	25 in-progress
Processor Resume Thresh.	24 in-progress

time to execute. (Cycle scaling is an ad-hoc technique of simulating superscalar systems. Three scaled cycles are equivalent to one “real” cycle, chosen to reflect the issue rate on a four-way superscalar machine. Timings are given in scaled cycles.)

A cached shared memory system is fully simulated; the protocol used is similar to the one described in [5]; sequential consistency is maintained, ignoring exo-ops. Memory access latency is 42 cycles, including directory manipulation, but not including the time needed to dispatch protocol messages. Memory banks have a one-block buffer, with a 12-cycle hit latency. The interconnection network is simulated at the packet-transfer level.

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 modules each having four banks of memory. Full-map cache directories are used; address space is organized into 16-byte blocks. At the CPUs, 2^{18}-byte, eight-way, set-associative caches are used. A list of simulation parameters appears in the table below.

Exo-ops with p operands take $p + 2$ cycles to issue, another 6 cycles to construct, and 21 cycles to execute each step. An exo-processor can start executing steps every 12 cycles. The additional time needed for cache and memory transactions and network transit time is fully simulated and part of operation timing. To limit congestion, processors with 25 in-progress exo-ops stall, resuming at 24.

4.2 Programs Tested

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. The fragments and their performance on the base system are discussed below; their performance on other configurations are discussed in Section 4.4. Also tested is a radix sorting program based on the radix sort in the SPLASH-2 suite [31].

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 two ways: using an atomic fetch-and-add instruction that executes at the CPU and exo-operations. In the histogram fragment conventional techniques are particularly inefficient and so the exo-op code is much faster. Of course, the code fragment is for comparison to systems which *do not* have increment-and-fetch instructions executed at memory.

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. A 1600-bin histogram was computed, elements were randomly distributed over bins with a uniform distribution.

Execution time for the fetch-and-add implementation was 156 cycles per iteration. The exo-operation implementation required only 17.1 cycles per iteration, over nine times faster. The conventional implementation suffers from false sharing and block contention on writes, the exo implementation not only avoids these but is non-blocking.

The nested-loops fragment (the conventional parallel implementation is shown below) computes a new array using two values from the old (new in the previous iteration) array. Symbols `A` and `B` are the base of two arrays, `perm` 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 `perm`) 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.

Code Fragment 1

```
for( outer = 0; outer < outer_end; outer++ ){
  /* Return a permutation. */
  int *perm = perms[outer];
  if( outer & 0x1 ){
     x=A; y=B; /* Odd iterations. */
    } else {
     x=B; y=A; /* Even iterations. */
  }
  barrier();
  for( inner = start; inner < stop; inner++ )
     x[inner] =  y[inner] + y[ perm[inner] ];
}}
```

The exo-op version uses a single exo-operation for the inner loop body with tags and with three parameters, `&x[inner][offs]`, `&y[inner][offs]`, and `&y[ p[inner] ][offs]`, in which `&base[index]` indicates the address of `base[index]`, and `offs` is `outer % 5`. (To reduce the cost of avoiding deadlock, storage is provided for several iterations, `offs` indicates which storage is used.) 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 216 instructions per iteration; the exo-op implementation takes 84.8, about 2.5 times faster.

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 75.8 and 19.7 cycles per iteration, respectively; exo-operations took 38.8 cycles per iteration, respectively. The exo-op approach is almost twice as fast as the conventional code that misses, however longer lines would reduce the exo-op version's advantage. When accesses hit, the conventional approach is faster, as one would expect.

Two permutation code fragments are simulated. The *in-place* fragment permutes some array elements (leaving the others unmoved); 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).

The conventional in-place permutation took 332 cycles per iteration; using exo-operations, 78.2, over four times faster. The copy permutation took 107 and 35.7 cycles, respectively; almost three times faster. By avoiding false sharing and coordinating writes, exo-ops perform the in-place permutation well. Exo-ops also improve the copy permutation.

4.3 Benchmark

Simulations were performed using a modified version of radix, a kernel from the SPLASH-2 benchmark suite [31] that implements a radix sort. Suggestions on placement of tasks and allocation of shared memory appearing in the benchmark were followed so that 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 to 524,288, sorted using radix 1024.

Radix uses exo-ops 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). Radix 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 3.53 million cycles; using exo-ops, 2.50 million cycles, a reduction of over 29%. The conventional code spends about 47% of its time ranking and 53% permuting. Exo-ops reduce the permute time by about a third, improvement is limited by read misses to the element being moved. Using exo-ops to permute avoids remote misses when computing the local histogram (using conventional instructions) in the following iteration, another benefit. Prefix time is

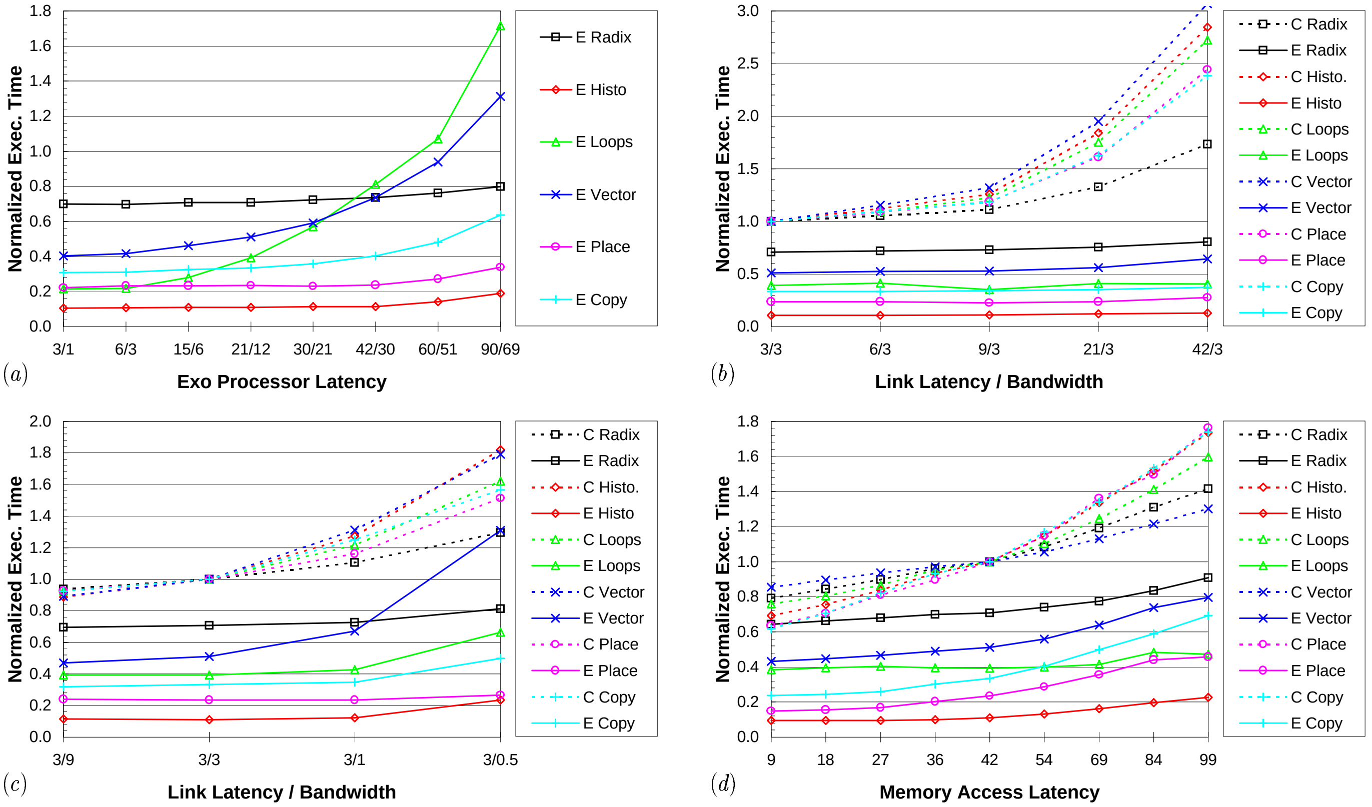

Figure 4. Execution time of programs normalized to base-configuration conventional systems (*e.g.*, 0.5 indicates half the execution time of a conventional system). Letter before program name indicates *Exo* or *Conventional* version. Parameters varied are (*a*) exo-processor latency/issue period (cycles), (*b*) network link latency (given as cycles of latency / bytes per cycle of bandwidth), (*c*) network link width (same format as latency), and (*d*) memory latencies (cycles).

a small fraction of execution time, so its reduction had little impact.

4.4 Configuration Effects

Because exo-ops are nonblocking system performance is determined by the rate at which they finish execution. Call the rate at which exo-ops finish execution the *completion rate* and the time from issue to completion the *completion latency*. Clearly when the completion rate is higher than issue rate, performance is insensitive to factors that affect completion rate. When completion rate is lower than the issue rate, CPUs will stall as the number of in-progress exo-ops exceeds the stall threshold. Thereafter, execution rate is sensitive to whatever is limiting completion rate.

Completion rate can be determined by the worst of network characteristics, memory latency, and exo-processor speed. Completion latency is determined by the sum of these, and also by the amount of time waiting for tags. Note that if a large amount of time is spent waiting for tags, the resulting completion latency will limit completion rate even though the memories, exo-processors, and network can keep up. (The stall is necessary to avoid the packet buffer overflow.)

In the simulation experiments, network, memory, and exo-processor speed were varied to produce these limiting effects. In the base system memory is usually the bottleneck, two-step exo-ops can be issued at a maximum of six times faster than memory accesses can be completed (accounting for the four banks). The maximum issue rate is 2.86 times what the network interface can handle (including all traffic generated by the exo-packet) and 1.71 times what the exo-processor can handle. (The only code that could achieve the maximum issue rate consists of uninterrupted—not even by a branch—exo-op issue instructions.)

In the first set of experiments exo-processor speed was varied. Normalized execution time of radix and the fragments is plotted in Figure 4(a) for a variety of exo-op execution speeds. Key x/y indicates x-cycle latency and y-cycle period (1/rate). Execution time is normalized to conventional code running on the base system. (Conventional programs are not shown on this plot. They are easily spotted on plots where they do appear since they all have an execution time of 1 at the base system value.)

Radix uses exo-ops for only a fraction of its execution time, and issues them at a lower rate, so it is less sensitive than the others. Its execution time barely improves below the base value of 21/12, and barely degrades even at 90/60.

Nested loops, in contrast is strongly affected. Each of the three steps in the exo-ops issued by the nested loops program must check tags; an exo-op step with an unsuccessful tag check takes additional time (a bandwidth effect) and also suffers higher latency. In the simulated system, any change to a block with waiting exo-ops, regardless of address, will cause the exo-processor to check the tags for those exo-ops, inflating the number of tag checks. For these reasons the nested loops fragment is most sensitive to exo-op speed. Also having three steps, the vector fragment is affected by exo-processor speed, but because it consumes less exo-processor bandwidth and does not use tags, it is less sensitive. With a single operand, histogram is least sensitive, suffering only when exo-processor speed replaces memory speed as the bottleneck. The in-place permutation is a special case since, to avoid deadlock, it never issues enough exo-ops to stall.

The effect of latency and bandwidth is plotted in Figure 4 (b). Axis label x/y indicates a link latency of x cycles and a bandwidth of y bytes per cycle. The plot shows the effect of increasing network latency while holding bandwidth constant (the network links act as deep pipelines or transmission lines). Increasing latency is disastrous for conventional programs, but has almost no effect on the exo systems, which are performance limited by various other factors discussed here. Shrinking network bandwidth, shown in Figure 4 (c), does impact performance. Vector is most affected since its exo-packets have three steps and is not limited by exo-processor speed as is nested loops. Also note that the conventional programs don't fair nearly as badly as they did with high-latency networks, nevertheless, exo systems' relative performance still increases as bandwidth goes down.

Memory effects are seen in Figure 4 (d);. Memory access latency is varied from 9 to 99 cycles, while protocol processing time is held constant at 12 cycles. As expected, conventional programs suffer while the effect on exo programs depends on their resource usage. As stated above, memory access time is a bottleneck for the exo programs, except for loops for which the exo-processor limits performance. This can be seen in the figure, as loops is least affected by changes in memory speed. Lower memory speed does not yield a dramatic improvement because of other bottlenecks.

The experiments described above show how resource limits affect performance. Also of interest is other factors affecting execution, line size, cache size, and system size. Line size primarily affects the conventional programs and those waiting for tags. Longer lines improve performance where there is spatial locality, worsening performance otherwise. Radix suffers both effects: the conventional and exo versions are fastest with 32-byte lines, performance degradation with larger lines is much worse in the conventional implementation. 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.

Reduced cache size slows down the conventional implementation of all of the fragments, and the exo implementation of radix, which also uses conventional memory access. The net effect is that exo-ops are of greater benefit on small-cache systems with the code fragments, but of a reduced benefit with radix, which is of course more important since it is more representative of real programs. Because of latency hiding, exo systems with more processors exhibit greater speedup over conventional systems.

5 Related Work

Exo systems are similar to many existing and proposed schemes that hide or avoid latency due to memory access and synchronization. Some schemes are similar in that the CPU issues short operations to remote systems. In smart memory schemes those operations execute on a memory or protocol processor, in some active message schemes they execute on a remote CPU. Some schemes are similar by exploiting proximity (on the same chip) to memory. Memory might share a chip with a functional unit capable of simple vector operations or a complete processor. Some schemes are similar in their use of efficient synchronization and latency hiding mechanisms. Active message multithreading systems effectively hide memory access latency and may be used to implement the tagged access descried here.

5.1 Remote Operations

Some smart memory and active message systems support short remote operations intended for computation. Smart memory is usually proposed to perform some amount of computation; proposals vary on the granularity and intent of the computation, from simple operations on single-bit operands, to synchronization support, to implementation of entire data structures [24].

Systems using smart memory for synchronization include the Cray T3E [26], a shipping commercial system, and Cedar [18], an older research system in many ways similar to the T3E. The smart memory implements atomic operations such as compare-and-swap, fetch-and-increment, and test-and-add.

In early work in this area, described by Stone [29], arithmetic and logical operations would be performed on cached data by a *logic-in-memory* cache. 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 [24]. 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].)

Multiprocessor designs might include a protocol processor to process messages directed at memories, for example as part of a directory-based coherence scheme [14]. Such a processor could also be used to implement something like exo-ops.

In active-message systems some messages can be assembled, dispatched, 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 [7,22,10] for active message implementation for the J- and M-machine [14] for FLASH, and [15] for EM-X.

5.2 Remote Operation Discussion

Exo-ops offer greater flexibility than systems providing atomic operations and the simple smart memory operations described above. Although systems providing protocol processors could implement exo-ops, the execution time would be much larger than that achievable on the specialized exo-processor. The same is true for active message systems, which also slow down the target CPU. Exo systems are designed so that exo-packets can be processed as fast as the network interface can deliver them. Any implementation in which a processor had to save and restore registers, etc. would not be able to keep up.

The active message machines are usually designed to support a multithreaded (e.g., M-Machine) or dataflow (e.g., *T) execution paradigm, different from each other and different from the nonblocking remote operations used by exo systems. Each may be better than the others at extracting parallelism on particular types of problems.

5.3 Memory-Processor Proximity

Recently, there has been much interest in placing a processor on the same chip as DRAM. Such an arrangement would not suffer chip-crossing delays and could make use of the large word sizes naturally available in memory designs. In some cases the processors are small, as with the smart memory schemes discussed above. In others a complete processor, the main computing resource, shares the chip along with a cache matched to the DRAM structure. (See [16,25].) With the program and data present on the chip, such a system has been shown to outperform a conventional processor/cache(s)/memory organization, even while using a less capable processor [25]. However, all but the smallest multiprocessors would require several chips, so this integration does not reduce communication-related delays.

It is natural to compare a system using such chips to one using exo-processors, since both have integrated processors and memory. An exo-processor is designed to execute packets only as fast as the network interface can provide them, divided by the number of banks. Thus smaller-area non-pipelined functional units can be used. Elaborate hardware needed to sustain multiple issue while faced with data dependencies is not needed. Exo-processors do need an exo-packet buffer large enough to support round-robin allocation, nevertheless their total size should be smaller than a conventional processor capable of any speed.

5.4 Latency Hiding

Latency hiding can be achieved by low-latency (e.g., simultaneous) multithreading, out-of-order issue perhaps combined with a relaxed consistency model, and prefetching.

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. Low-latency 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 [9] for an early description and [1,2,8,23,30] for some recent work.)

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].

In *prefetching*, data is moved into a cache before needed. Prefetching can be accomplished by having the programmer or compiler insert prefetch instructions for data ahead of its use [11,21,26], 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) [6,26], or, simplest of all, by using long cache lines.

5.5 Latency Hiding Discussion

Each of these approaches is quite different in its form of parallelism. Out-of-order issue is most similar to exo systems in that the thread issuing instructions needing remote data does not block until the computed data is needed. The instructions remain in the processor, executing when the data arrives. It is also the least practical since no reasonable processor could hold instructions for the hundreds of cycles that might be needed to service a shared cache miss.

With prefetching the programmer, compiler, or hardware must determine memory access addresses in advance of their need, which is not always possible.

With low-latency multithreading the programmer or compiler must provide multiple threads per processor. The system is kept continuously busy only if enough threads are available. Dividing a program into additional threads will typically add overhead, whereas exo systems need no more than one thread per processor.

5.6 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 [19,27]. 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 [23]. 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 Conclusions

A parallel system where memory modules can execute simple operations using an exo-processor has been presented. Advantages include avoiding cache misses and pollution, avoiding contention, and tight synchronization. Their effectiveness on a radix sorting program and some code fragments was demonstrated through execution-driven simulation. The radix program takes 29% less time. The speedup on the code fragments (which do not indicate whole program performance) was much higher, usually over twice as fast.

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. In ongoing work, existing programs are being adapted to exo-ops and algorithms which can make use of exo-ops are being implemented. Exo-op execution, because it can wait for memory, can be something like execution on a dataflow machine. Nevertheless, programs adapted for exo systems will be written in procedural languages (presently C). Procedural languages are being used because they are well-established (no additional time needed to develop or adapt compilers) and programs in these languages execute efficiently.

7 Acknowledgments

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

8 References

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

[2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, "The Tera computer system," in *Proc. of the Int. Conf. on Supercomputing*, June 1990, pp. 1–6.

[3] E.A. Brewer, C.N. Dellarocas, A. Colbrook, and W.E. Weihl, "Proteus: a high-performance parallel-architecture simulator," in *Proc. of the ACM SIGMETRICS Conf.* May 1992.

[4] D. Burger, J.R. Goodman, and A. Kagi, "Memory bandwidth limitations on future microprocessors," in *Proc. of the 23rd Int. Symp. on Computer Arch.* May 1996, pp.78–89.

[5] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal, "Directory based cache coherence in large–scale multiprocessors," *IEEE Computer*, vol. 23, no. 6, pp. 49–59, June 1990.

[6] T. Chen and J.. Baer, "Effective hardware-based data prefetching for high-performance processors," *IEEE Trans. on Computers*, vol. 44, no. 5, pp. 609-623, May 1995.

[7] W.J. Dally, J.A. Stuart Fiske, J.S. Keen, R.A. Lethin, M.D. Noakes, P.R. Nuth, R.E. Davidson, and G.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.

[8] J.B. Dennis, G.R. Gao, and R.A. Iannucci (Editor), "Multithreaded computer architecture," Boston: Kluwer Academic Publishers, 1994, Chapter 1, pp. 1–72.

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

[10] M. Fillo, S.W. Keckler, W.J. Dally, N.P. Carter, A. Chang, Y. Gurevich, and W.S. Lee, "The M-machine multicomputer," in *Proc. of the 28th Annual Int. Symp. on Microarchitecture,* Nov. 1995, pp. 146–156.

[11] K. Gharachorloo, A. Gupta, and J.L. Hennessy, "Two techniques to enhance the performance of memory consistency models," in *Proc. of the Int. Conf. on Parallel Processing,* August 1991, vol. I, pp. 355–364.

[12] M. Gokhale, B. Holmes, and K. 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 Arch. News,* vol. 19, no. 3, pp. 254–263, May 1991.

[14] J. Heinlein, K. Gharachorloo, S. Dresser, and A. Gupta, "Integrating of message passing and shared memory in the Stanford FLASH multiprocessor," in *Proc. of the Int. Conf. on Architectural Support for Programming Languages and Operating Systems,* October 1994, pp. 38–50.

[15] Y. Kodama, H. Sakai, M. Sato, H. Yamana, S. Sakai, and Y. Yamaguchi, "The EM-X parallel computer: architecture and basic performance," in *Proc. of the Int. Symp. on Computer Arch.* June 1995, pp. 14–23.

[16] P.M. Kogge, "Execube—a new architecture for scalable MPPS," in *Proc. of the Int. Conf. on Parallel Processing,* Aug. 1994, vol. I, pp. 77–84.

[17] D.M. Koppelman, "Ver. L3.10 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.

[18] 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 *Proc. of the Int. Symp. on Computer Arch.* May 1993, pp. 213–223.

[19] B. Lee and A.R. Hurson, "Dataflow architectures and multithreading," *IEEE Computer,* vol. 27, no. 8, pp. 27-39, Aug. 1994.

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

[21] T.C. Mowry, M.S. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," in *Proc. of the Conf. on Architectural Support for Programming Languages and Operating Systems,* October 1992, pp. 62–73.

[22] M.D. Noakes, D.A. Wallach, and W.J. Dally, "The J-machine multicomputer: an architectural evaluation," in *Proc. of the Int. Symp. on Computer Arch.* May 1993, no. 20, pp. 224–235.

[23] R.S. Nikhil and G.M. Papadopoulos, "*T: a multithreaded massively parallel architecture," in *Proc. of the Int. Symp. on Computer Arch.* May 1992, pp. 156–167.

[24] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A case for intelligent RAM," IEEE Micro, March 1997, vol. 17, no. 2, pp. 34–43.

[25] A. Saulsbury, F. Pong, and A. Nowatzyk, "Missing the memory wall: the case for processor/memory integration," in *Proc. of the 23rd Int. Symp. on Computer Architecture,* May 1996, pp. 90–101.

[26] S.L. Scott, "Synchronization and communication in the T3E multiprocessor," in *Proc. of the Int. Conf. on Architectural Support for Programming Languages and Operating Systems,* October 1996.

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

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

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

[30] D.M. Tullsen, S.J. Eggers, J.S. Emer, H.M. Levy, J.L. Lo, and R.L. Stamm, "Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor," in *Proc. of the Int. Symp. on Computer Arch.* May 1996, pp. 191–202.

[31] S. Cameron Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta, "The SPLASH-2 programs: characterization and methodological considerations," in *Proc. of the Int. Symp. on Computer Arch.* May 1995, pp. 24–36.