Achieving Latency Tolerance and Fine-Grain Synchronization Using Externally Executed Memory-to-Memory Instructions in Multiprocessors

David M. Koppelman*
102 EE Building, Department of Electrical & Computer Engineering
Louisiana State University, Baton Rouge, LA 70803
Voice: (504) 388-5482, Fax: (504) 388-5200, E-mail: koppel@ee.lsu.edu

Abstract: A multiprocessor using new types of instructions is described and analyzed. The instructions, called *exo-scalar*, are issued in the usual way but are executed externally, at caches or memory units. Exo-scalar instructions can specify three-operand arithmetic operations and memory reads and writes; the operands can be memory addresses or immediate data; the steps needed to carry out an instruction may need to wait for, and can result in the changing of, tag bits associated with a memory location. The tag bits can be used to insure proper access ordering and to implement synchronization operations. Although issuing an exo-scalar instruction can take many cycles, for example to move operand memory addresses from CPU registers, overall execution time is reduced because stalls for long-latency memory or synchronization operations are less frequent. When memory consistency is needed processors can wait for a counter tallying in-progress exo-scalar operations to return to zero. Because of implementation simplicity and ease of extracting parallelism, exo-scalar operations are better at achieving fine-grain parallelism than active messages, dataflow, and multithreading. The effectiveness of exo-scalar instructions was evaluated using trace-driven simulation. Kernels from the Splash 2 benchmark suite were run on simulated systems with and without exo-scalar operations: reductions in execution time of 25% and more were observed.

1 Introduction

Performance of coherent cache shared-memory parallel computers, multiprocessors, can fall below their potential maximum because of memory-access latency and tight synchronization requirements. Further, the instruction-level parallelism techniques for improving single-processor performance, pipelining and superscalar issue, are at a point where large increases in hardware complexity are needed to realize even small performance improvements. Access- and synchronization-related performance degradation can be avoided and additional instruction-level parallelism can be extracted by using the technique described here, called *exo-scalar*. In an exo-scalar processor, part of a coherent cache shared memory parallel computer, *exo-scalar instructions* are issued within a processor but are completed outside the processor at an *exo-scalar execution unit* (EEU). The EEU may be located near the processor, near a memory module, or near both. The number of operands can vary and can be immediate or indirect. (Because operand data and addresses are copied from the processor, issue time is larger than individual RISC-type instructions. Since the amount of data

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

transfered by an exo-scalar instruction is no more than the load and store instructions it replaces there is no need for additional bandwidth at the processor/bus interface.)

Processors normally don't stall waiting for a just-issued exo-scalar instruction to complete, nor do they need to store much state information for in-progress exo-scalar instructions: a single counter suffices. Exo-scalar instructions signal their completion by sending an acknowledge to the issuing processor, which would decrement this counter. When necessary, the processor can wait for the count to reach zero.

Exo-scalar instructions specify arithmetic and load/store operations; they also specify where the operations are to be carried out: *cache-side* or *memory-side*. Cache-side instructions' read and write operations make use of the issuing processor's cache. Since lines brought in by the instruction can be read by the processor and *vice versa*, processor locality is exploited (or confounded, if needed lines are evicted, in which case memory-side instructions might be the better choice).

Memory-side instructions are sent to the memories in which their operands reside, making several trips when their operands are on more than one memory. Unlike cache-side and conventional memory-access instructions, memory-side instructions do not bring a line into the issuing processor's cache, which is desirable when data is to be read just once. When a line accessed by a memory-side instruction is not cached a write can proceed without delay, possibly avoiding the false sharing [11] that would occur when a line is written by several processors.

Memory-side instructions can be used for synchronization, much tighter synchronization than could be achieved using conventional methods. Memory locations or data types are associated with a *state tag*, a field a few bits long read and written by exo-scalar instructions. These are similar to, but more general than, the full/empty presence bits used in HEP, Tera, and other machines [2,9]. An exo-scalar instruction ready to access a memory location might wait for the location's state tag to be set to some value; as a result of accessing the location, the exo-scalar instruction might change a state tag.

State tags can be used, for example, to permute an array in place. (That is, rearrange elements of array a so that for all indices i the element at position i is moved to position $\pi(i)$, where π is a permutation of the array indices.) Initially, all state tags are set to 0. A memory-side exo-scalar instruction reads element a_i and sets its state tag to 1; it proceeds to the memory in which $a_{\pi(i)}$ is stored and writes the element after the tag associated with the memory is set to 1. State tags can also be used with exo-scalar instructions specifying arithmetic operations.

1.1 Alternate Approaches to Latency Hiding

The latency tolerance, parallelism, and synchronization achieved in exo-scalar processors is less efficiently realized using existing techniques. Latency tolerance in multiprocessors can be achieved using out-of-order completion combined with a relaxed consistency model, multithreading, and prefetching.

In a system using out-of-order completion and a relaxed consistency model, instructions following memory accesses which do not immediately complete (*e.g.*, due to a cache miss or some consistency action) would not necessarily stall [12,13]. Memory writes (stores) 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 (loads) 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 processor 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, an exo-scalar processor issues memory reads, arithmetic operation(s), and writes in a single unit to be processed elsewhere; the number of outstanding exo-scalar 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 [12,16,19,21], 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], or, simplest of all, by using long cache lines (for when memory locations are sequentially accessed). For all of these techniques the memory location must be determined in advance of need, which is not always possible. In contrast, exo-scalar operations are issued when the address are known.

An alternative to executing past a stalled instruction or predetermining an address is 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 can be performed very fast (since they don't have to save and restore registers, for example). (See [10] for an early description and [1,2,18,9,22,25] for some recent work.) While latency is hidden, the processor 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-scalar system is outside the processor, which may be a desirable tradeoff.

1.2 Alternate Approaches to Parallelism Extraction

Exo-scalar systems can extract additional parallelism, even when memory-access latency is small. This additional parallelism could be obtained on non-exo-scalar systems by adding additional processors or using exotic techniques such as dataflow.

Adding processors increases the cost of the system and often reduces efficiency. The cost includes the processors themselves and the additional network ports needed to connect them.

Exo-scalar systems also have additional processors, the EEUs, but these are much simpler. They do not require the register sets or the complexity needed to achieve superscalar execution (since operations could be issued no faster than the exo-scalar packets could be constructed and delivered). The EEU at a memory only accesses that memory, and so an elaborate coherent memory interface is not needed. In fact, the EEUs might be implemented by the network or memory processors that some systems already provide.

Some proposals employ execution schemes which are similar to exo-scalar instructions. Horst describes a *task flow architecture* in which instructions are stored adjacent to data storage in a *memory packet* [15]. A memory packet also includes a pointer to the next memory packet to execute and something like a state tag. Execution proceeds by sending data between memory packets. In such a system instruction execution may include the overhead of interconnection network traversal, and even if such instructions exhibit processor locality it is far more difficult to realize the efficiencies of a conventional processor: fast instruction issue into a pipelined execution unit accessing high-speed registers. In comparison, an exo-scalar system would only issue exo-instructions when they would be of benefit. Also, memory locations are not associated with a fixed exo-instruction, and an exotic programming paradigm is not needed.

Dataflow is perhaps the ultimate way to extract parallelism. "Classical" dataflow machines have been inefficient [18,23], efficiency has since been improved using designs more like conventional processors. One such machine is *T, which is designed so that tasks can be stopped and started quickly when needed remote data arrives [22]. 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. Exo-instructions, in contrast, do not explicitly restart a task and may perform several actions; *T is to be a hybrid dataflow machine (simulation results were not reported), whereas the scheme described here is much closer to a conventional processor.

In general dataflow, considered by many to be a compelling but impractical idea, is hobbled by programmers' difficulty in writing dataflow code and in the inefficiency of dataflow hardware. Exo-scalar instructions that use state tags execute in a similar fashion to dataflow operations: they proceed to functional units and execute when all operands are available, perhaps triggering additional operations. The major difference is that exo-scalar operations are emitted by procedural code and so the programmer does not have to cast the application's control flow into some data-flow paradigm.

Exo-scalar instructions are in some way similar to certain smart memory schemes. In these memory units have some computational capability, simple operations performed in parallel on large sets of data or complex operations on smaller sets of data. Examples of the former include image-processing type operations. Such systems would include many inexpensive memory processors which when used, would outperform the much smaller number of main processors; they

would be inexpensive enough so that low overall utilization would be acceptable. The parallelism extracted here is complementary to that extracted using exo-scalar instructions.

In other smart memory schemes the memory would implement data types and 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 main processor.

1.3 Other Related Work

Active messages might be used to implement something similar to exo-scalar instructions. 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 and [14] for FLASH. While the issuing of an exo-scalar instruction is almost identical to the sending of an active message, the interrupt of the processor at an active message's destination introduces latency and processor overhead that exo-scalar operations avoid. If a context switch is needed then many cycles are lost—both by the active message and the interrupted task. Exo-scalar operations, in contrast, can be handled by a simple execution unit which includes no more context than is present in the exo-scalar packet and the memory location read, eliminating the need for state saving.

The remainder of the paper is organized as follows. Terminology and references on multiprocessors are provided in Section 2 [this page]. Exo-scalar processor details are presented in Section 3 [next page]. The methodology of the trace-driven simulation evaluation appears in Section 4 [p. 7] and results are described in Section 5 [p. 9]. Conclusions appear in Section 6 [p. 12].

2 Preliminaries

The following terminology will be used. The systems discussed consist of N *processors* interconnected by some interconnection network. Each processor is associated with a *memory module* and a *cache* which are collectively referred to as the *memory system.* All *tasks* in a parallel program share the same *address space;* the memory system manages the address space in units called *blocks.* The caches are a-way set-associative; each set holds a *lines.* A line holds exactly one block. *Line* will be used to refer to cache storage while *block* will refer to the portion of the address space. (*E.g.*, a line could hold many different blocks.) Blocks are *mapped* to memory modules. The memory module to which a block, or an address in the block, is mapped is called its *home memory.* Cache *coherence* is maintained using a directory-based write-invalidate protocol. For general information on coherent caches see [5,20,24].

3 The Technique

The elements of an exo-scalar instruction appear below; possible instruction formats are discussed later.

$$\begin{array}{ll} \text{EXO}-\text{OP} & \langle\text{Mode}\rangle\ \langle\text{Data}\rangle \\ & \langle\text{Pre}_1\rangle\ \langle\text{Opcode}_1\rangle\ \langle\text{Operand}_1\rangle\ \langle\text{RW}_1\rangle\ \langle\text{Post}_1\rangle \\ & \langle\text{Pre}_2\rangle\ \langle\text{Opcode}_2\rangle\ \langle\text{Operand}_2\rangle\ \langle\text{RW}_2\rangle\ \langle\text{Post}_2\rangle \end{array}$$

A particular instruction can use some or all these fields. Field $\langle\text{Mode}\rangle$ indicates if the instruction is cache side or memory side. Field $\langle\text{Data}\rangle$ holds the data copied from the issuing processor's registers; field $\langle\text{Operand}_1\rangle$ specifies the memory location of operand 1. Each memory location contains several bits of state tag; this may or may not be part of the memory location, depending on implementation. Field $\langle\text{Pre}_1\rangle$ specifies the tag value that must be present at $\langle\text{Operand}_1\rangle$ before operation $\langle\text{Opcode}_1\rangle$ will proceed; after the operation, the state tags will be set to $\langle\text{Post}_1\rangle$. Null values can be specified for $\langle\text{Pre}_1\rangle$ and $\langle\text{Post}_1\rangle$ indicating that the state tags should be ignored and that the state tags should not be changed, respectively. Field $\langle\text{RW}_1\rangle$ indicates whether the result of the operation should be written back to $\langle\text{Operand}_1\rangle$ or should replace $\langle\text{Data}\rangle$ (within the exo-packet). The fields for the second operation are correspondingly defined.

An actual instruction format that included all of these fields would either be larger than the 32 bits used by most modern architectures or would limit the field sizes. Alternately, the $\langle\text{Data}\rangle$, $\langle\text{Operand}_1\rangle$, and $\langle\text{Operand}_2\rangle$ fields could be complete (*e.g.*, five bits on a machine with 32 registers), while a single index is used to specify the other fields. The index would refer to an entry in a table which fully specifies the values of the remaining fields. The table, maintained by the EEU, is initialized by the processor. If the index field is large enough, the processor would rarely if ever need to write this table after initialization and so exo-instruction issue time is kept to a minimum.

As an example, the code below performs the in-place array permutation mentioned in the introduction, with the exo-instruction shown as a function call (to illustrate the field values).

```
#define Unaccessed 0
#define Accessed   1

for(i=start;i<end;i++){
 EXO_OP(Memory_Side, Null,
        Unaccessed,  Opc_NOP,  & a[i],         RW_Read,    Accessed,
        Accessed,    Opc_NOP,  & a[ pi[i] ],   RW_Write,   Unaccessed); }
```

Note that the arithmetic operations are no-ops, indicated by `Opc_NOP`. The time to complete an iteration of this loop includes the time to compute the operand addresses, the time to issue the exo

instruction, and the time to increment and test `i`. If read and write instructions were used instead (in which case the permuted array would use different storage than the original) their miss time would add considerably to the loop iteration time. Long cache lines would reduce the number of read misses, but could not reduce the number of write misses when `pi` is arbitrary. See Section 4.2 [next page] for additional examples.

3.1 Completion Counter

Each processor maintains a count of outstanding exo-scalar instructions that it had issued and has a mechanism allowing it to wait until the counter reaches a certain value. For consistency purposes the processor might wait for the counter to reach zero; to keep other parts of the system from being overloaded the processor might stall if the counter exceeds some threshold. The former would be done using an explicit wait command, the later would be automatic.

3.2 Hardware Outline

Details of an exo-scalar implementation are outlined below; specifics of the simulated systems are in Section 4 [this page]. An exo-scalar processor, in addition to the usual multiprocessor hardware, would include EEUs and exo-packet storage at each memory and cache. The storage used for cache and memory information (state, etc.) would include a pointer into the exo-packet storage.

Upon issue the EEU would construct an exo-packet in the exo-packet storage. After construction packets for memory-side instructions are sent through the network to the appropriate memory unit; the EEU initiates a cache access for cache-side packets. If a cache-side access misses a pointer to the packet is placed on a list of non-cached packets, if the access hits but the line is not in the needed state the packet is added to a list associated with the line. Otherwise, the EEU performs the operation on the cache line and then initiates the next operation (specified in the exo-packet), either another cache access or an acknowledgment.

When a cache line undergoes a state change, such as when an exclusive state is granted by memory, the line's exo-packet pointer is tested and if non-null the packets are checked to see if execution can proceed. Note that most lines would have null pointers, and that for well-written code, most non-null lists would be of length one.

Memory-side exo-packets are sent to the location of their operands. The actions taken by the EEU at the memory are similar to those taken by the cache's EEU.

4 Methodology

The effectiveness of the technique was evaluated by simulating the execution of some Splash 2 kernels on systems with and without exo-scalar processors. A modified version of Proteus, a trace-driven parallel computer simulator, was used [4]. The simulator provided execution time and other data for a variety of system configurations.

4.1 Simulator

The execution of a program on a multiprocessor is simulated in Proteus using an augmented copy of the program (a copy in which small segments of code are inserted) compiled for the host machine (the machine the simulator runs on). The inserted code keeps track of simulated time and branches to simulator code when appropriate. Simulated program instructions that do not access memory take one cycle of simulated time to execute, see Section 4.3 [next page] for timing details on the other instructions. A coherent cache shared memory system is fully simulated; the protocol used is similar to the one described in [17]. Sequential consistency is maintained, except of course for locations accessed by exo-instructions. The interconnection network is simulated at the packet-transfer level.

The version of Proteus used for this study is a modified version (in addition to the exo-scalar modifications) of Proteus Version 3.1. Relevant modifications are in the handling of shared memory and compiled code. In the modified version, dynamic memory allocation can provide a contiguous block of memory addresses divided between the memory modules of all processors. (In version 3.1 a single allocation could reside on only one processor.) 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 victim is chosen randomly. For details on modifications see [17].

4.2 Benchmarks

Simulations were performed using two kernels in the Splash 2 benchmark suite [26], LU, an LU factorization program, and Radix, a radix sorting program. [Note to reviewers: more benchmarks may soon be tested]. The programs contain suggestions on placement of tasks and allocation of shared memory. These suggestions were followed so that the programs were well tuned, with or without exo-scalar instructions. The benchmarks were compiled with optimization on and run using the base problem sizes specified in the code.

The benchmarks were modified so they could use exo-scalar operations. Radix uses exo-scalar operations to permute the keys which it is sorting and to compute a prefix sum. In the code permuting keys, memory-side exo-instructions avoid the write misses and false sharing otherwise encountered when a key is written to its new location. The prefix sum is computed using a linear chain of additions, each addition issued by a different processor. (A parallel prefix is not used because many prefix sums are needed; a parallel prefix, which takes less time but requires more operations, would only perform well for impractically small radixes and large systems.) The additions are performed by memory-side exo-instructions that perform an addition after the previous sum in the chain is completed. Using exo-instructions the prefix sum is computed using about one message per element whereas the conventional prefix algorithm would use 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.

In LU cache-side exo-instructions are used in a *daxpy* loop, a loop which computes `y[i] += x[i] * a`. The hit ratio for these exo-instructions is high, so their benefit is overlapping index computation with data computation rather than latency hiding.

4.3 Configurations

The simulated systems discussed in the next section are described in terms of differences with a base configuration, described here. The base system is a 16-processor system using a two-dimensional mesh interconnection network. The memory system consists of 16 2^{19}-byte memory units organized into 16-byte lines each having a ten-entry limited directory. (The alert reader will have noted that a full-map directory would use less memory; because the degree of sharing on the benchmarks tested is small the performance difference would be very insignificant.) At the processors, 2^{18}-byte eight-way set-associative caches are used. A list of simulation parameters appears in the table.

Simulation parameters relevant to exo-instructions were conservatively chosen. Exo-instructions take four cycles to issue, another five cycles to construct, and seven cycles for each operation. Additional time is added for cache and memory transactions and network transit time used. A processor with ten in-progress exo-instructions will stall; it resumes at nine. See the table for a list of parameters.

5 Results

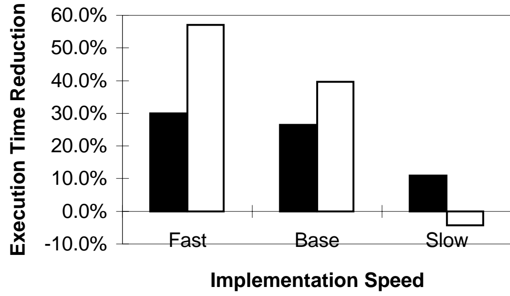
Simulations were run to find the performance improvement on a typical system and to determine the effect of system size, cache size, and network characteristics on performance. The impact of changes in exo-instruction issue, construction, and execute time were also evaluated.

The performance improvement achieved using exo-scalar instructions is plotted in Figure 1(a) for the base configuration and a fast and slow implementation. For the base configuration, there is a performance improvement of over 26% running Radix and almost 40% running LU. In the fast implementation, exo-instructions are issued in 2 cycles and exo-packets are constructed in another 2 cycles; execution takes 3 cycles. In the slow implementation, issue, construction, and execution take 6, 8, and 15 cycles, respectively. For Radix, the difference between the fast and base speeds are modest; in these runs exo-instructions are completed at the same rate they are issued. At the slow speed, exo-instructions are issued faster than they are executed and so their slower reduces their effectiveness. The loop issuing exo-instructions in LU is tighter than those in Radix, and only cache-side exo-instructions are used (no network delays), so the impact on performance is more direct. The fast implementation runs over twice as fast, while the slow implementation runs a bit more slowly.

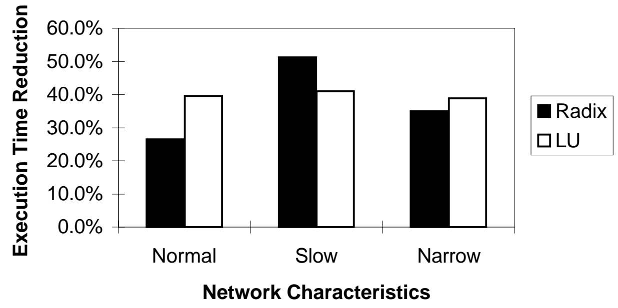
Because processors need not stall while using shared memory, exo-scalar systems' network traffic is higher. On severely bandwidth-limited systems (systems where increased traffic does not

Table 1: Parameters for Base Configuration

Simulation Parameter	Value
System Size	16 processors
Network Topology	$k^d = 4^2 = 4 \times 4$ mesh
Cache Size	$n_s = 2^{11}$ sets
Cache Associativity	8
Cache Line Size	$n_l = 16$ bytes
Cache Capacity	262,144 bytes
Cache Hit Latency	1 cycle
Memory Module Capacity	2^{19} bytes
Address Space Size	$A = 32$ bits
Directory Size	10 entries
Memory Latency	5 cycles
Protocol Message Size	$n_{pr} = 6$ bytes (plus data)
Network Interface Width	8 bytes
Network Link Width	$w = 2$ bytes
Network Wire + Switch Delay	3 cycles
Exo-Packet Size	$8 + 8d$ bytes
Exo-Instruction Issue Latency	4 cycles
Exo-Packet Construction Time	5 cycles
Exo-Instruction Execute Time	7 cycles (per operation).
Processor Stall Threshold	10 in-progress
Processor Resume Threshold	9 in-progress



(*a*)



(*b*)

Figure 1. Performance improvement of exo-scalar systems over conventional systems for several (*a*) implementation speeds and (*b*) network capabilities.

result in any increased throughput) execution time is dominated by data volume, so there is no advantage in issuing multiple requests without stalling since this would only fill an output queue. On systems that are moderately bandwidth-limited (message latency high compared to an lightly loaded network, but increased traffic results in increased throughput), exo-instructions should result in performance improvement by hiding latency. In systems having high latency and high bandwidth, exo-instructions should do best since network latency is hidden and the traffic can be handled.

These effects are illustrated in Figure 1(*b*), in which performance improvement in the base system, a high-latency system, and a low-bandwidth system are plotted. The high-latency system

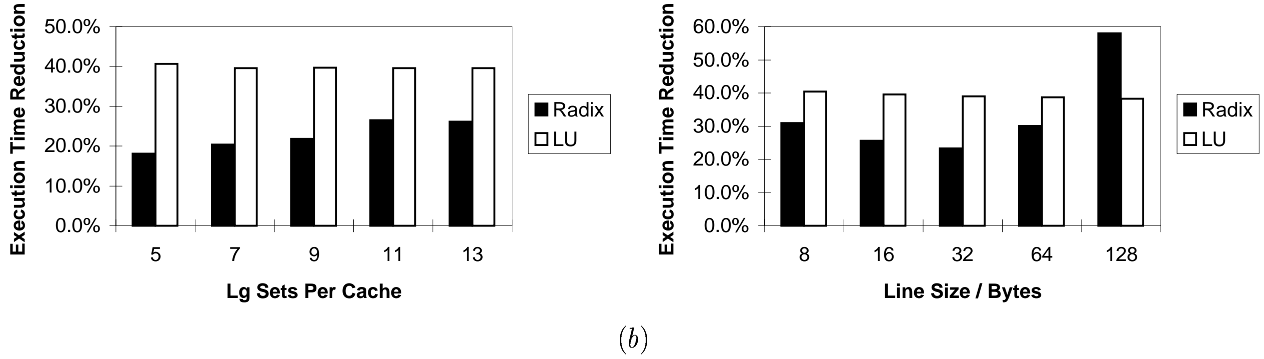


Figure 2. Performance improvement v. (*a*) cache size and (*b*) line size.

uses links with a 40-cycle delay, compared to 2 cycles in the base system. (The link delay, suffered only by the first packet in a message, can represent propagation delay as well as setup delays.) The link widths in the low-bandwidth system are 4 bits; the base system uses 16-bit links. As expected, the largest performance improvement is in the high-latency system where Radix runs about twice as fast; improvement is also better running LU. The relative performance of exo-scalar operations increases in bandwidth-limited systems running Radix, but for LU the base system yields a slightly higher performance improvement. LU's smaller changes are due to its use of cache-side operations, which generate the same traffic as conventional instructions, albeit at a faster rate.

Improvement versus cache size is plotted in Figure 2(*a*), where the number of sets is varied from 2^5 to 2^{13}. The performance improvement of Radix drops with small caches, this is due to cache misses in access to keys during the permute phase. (A digit in the chosen radix is used to determine where the key is to be moved.) There is little change in LU, where the hit ratio is high.

The effect of line size is plotted in Figure 2(*b*), where the line size is varied from 8 to 128 bytes, while keeping the cache size fixed at 2^{18} bytes, the capacity of the base system. The unmodified version of Radix does best with a line size of 32 bytes: less spatial locality is exploited with smaller line sizes, whereas larger line sizes increases the amount of false sharing. Most false sharing is avoided when exo-instructions are used, so performance is best at a higher line size. The relative performance of exo-instructions is highest at the largest line size, where false sharing in the conventional system reaches pathological levels. Comparing the minimum execution times, which is perhaps most fair, yields a benefit of 24.2%.

LU does not suffer from false sharing and by design exhibits excellent spatial locality so the line size effects on LU are less pronounced. The benefit varies from 40.5% to 38.3%, the benefit is 38.7% comparing minimum execution times.

The performance improvement on hypercube-interconnected systems is plotted in Figure 3. Running Radix, performance gains of 20 to 25% are obtained; running LU gains of 32 to 40% are obtained. Performance improvement in 4-processor Radix systems may be lower due to the uneven memory distribution in the exo-scalar version of Radix. Benefit in LU is close to 39% for all but one run; the cause of the anomalous point is unknown.

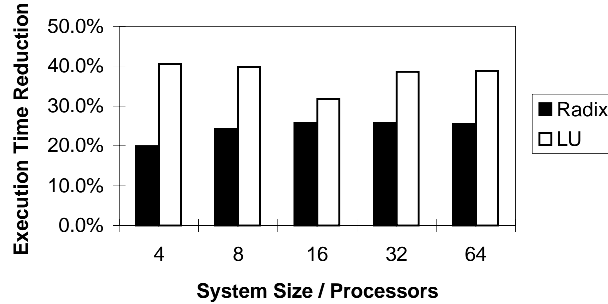


Figure 3. Performance improvement v. system size.

6 Conclusions

A multiprocessor in which instructions are issued within a processor but execute externally was described and analyzed. External execution has several benefits: fine grain parallelism is extracted, shared-memory-related latency is avoided, false sharing is avoided, and fine-grain coordination between processors is easily achieved (using state tags). The performance improvements of 25 to over 50% suggest that the technique can be effective. Other methods exist to extract similar parallelism, but none offer exo-scalar's flexibility without using exotic programming techniques or requiring many threads.

Further work is needed to determine how broadly exo-scalar instructions can be used, in particular, can exo-scalar execution benefit a significant fraction of problems for which multithreading and vector techniques are ineffective? An important avenue for research is in automatic insertion of exo-instructions. This would require knowledge of access patterns so that exo-instructions, which do not execute consistently, realize the desired read/write ordering. If EEU implementation cost is low compared to a memory module or cache, then multiple units could be provided, perhaps one for each bank in an interleaved memory module. The ultimate practicality depends upon implementation speed, the number of programs which can be sped up, and the ease of converting programs.

7 References

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

[2] Robert 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] Eric 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 Stenstroem, “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] Susan J. Eggers and Tor E. Jeremiassen, “Eliminating false sharing,” in *Proceedings of the International Conference on Parallel Processing,* 1991, vol. I, pp. 377–381.

[12] 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.

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

[16] Alexander C. Klaiber and Henry M. Levy, "An architecture for software-controlled data prefetching," in *Proceedings of the International Symposium on Computer Architecture,* May 1991, pp. 43–53.

[17] David M. Koppelman, "Version L3.7 Proteus Changes" (informally distributed documentation), soon available (an earlier edition currently available) at ftp://gate.ee.lsu.edu/pub/koppel/proteus/lsuProtDoc.ps

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

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

[25] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy, "Simultaneous multithreading: maximizing on-chip parallelism," in *Proceedings of the International Symposium on Computer Architecture,* June 1995, pp. 392–403.

[26] 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.