A Multiprocessor Memory Processor for Efficient Sharing And Access Coordination

David M. Koppelman
Department of Electrical & Computer Engineering
Louisiana State University, Baton Rouge

koppel@ee.lsu.edu http://www.ee.lsu.edu/koppel

Outline

- Multiprocessor Problems
- Smart Memory Solution
- Nested Loops Example
- Simulation Evaluation
- Related Work
- Feasibility
- Future Work

Presentation Note

Slides with two-level numbering will be covered if interest expressed.

Multiprocessor Problems

Miss Latency

Penalty can be 100s of cycles.

Substantial impact on some programs.

Synchronization Overhead

Barrier overhead possibly 1000s of cycles.

Locks, rendezvous, etc. also time-consuming.

As a result . . .

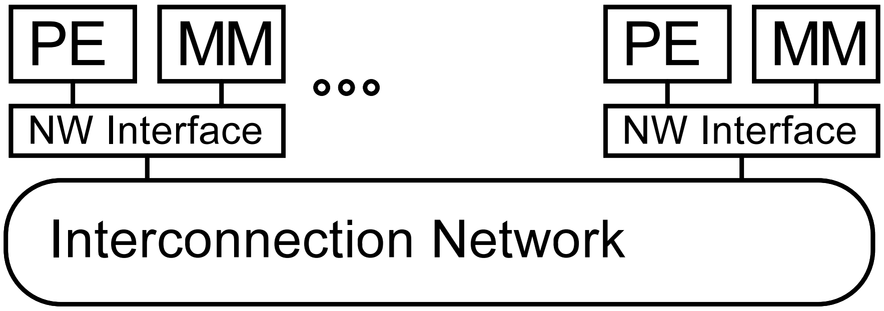
. . . some useful programs slowed . . .

. . . and many algorithms are unimplementably inefficient.

Exo-Processor System Solution

Derived From Typical Multiprocessor

- Processing elements mostly normal.
- Memory modules (of course) can process.



A sequentially consistent shared address space assumed . . .

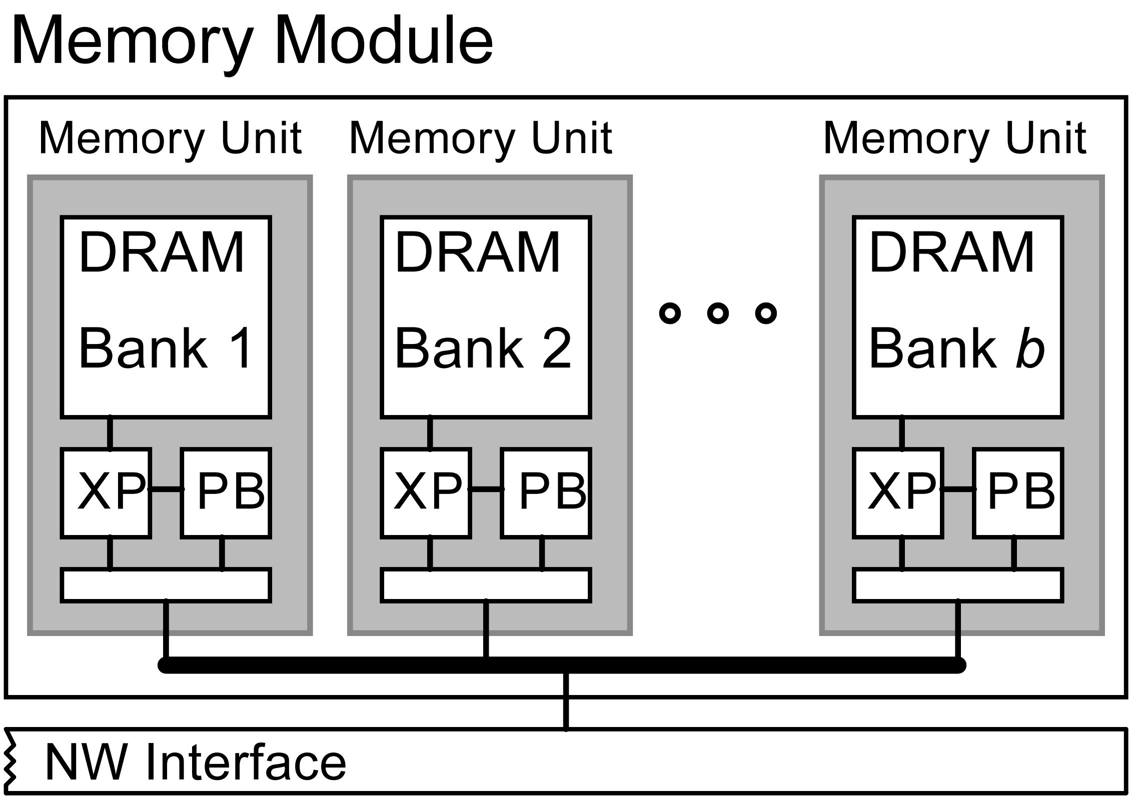
. . . but not required.

Processing Element

Processing Element Features

- Conventional processor (CPU).
- Very-low-latency message dispatch capability.

Memory Module



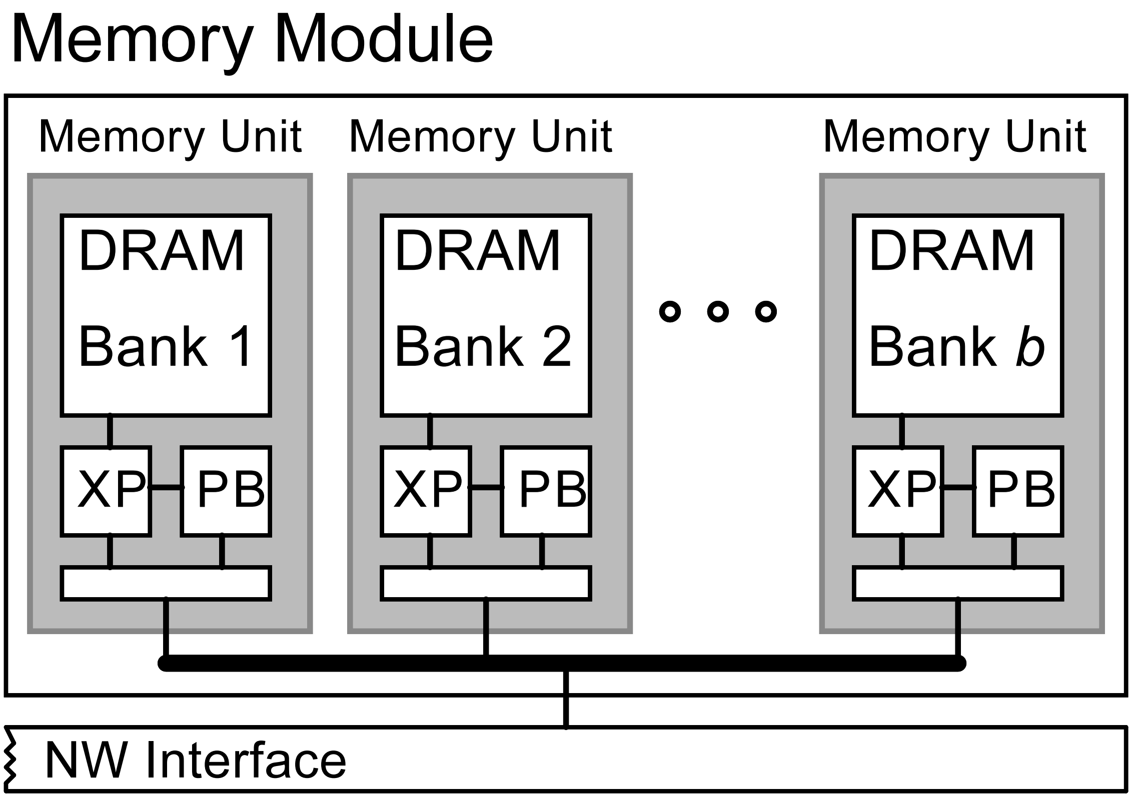
Memory Module Features

- Uses single network interface (NWI).
- Contains multiple *memory units*.

Memory Unit Features

- DRAM, implements address space.
- *Exo-processor* (XP) controls unit.
- *Packet buffer* used by XP.

Exo-Processor



Exo Processor Capabilities

- Executes context units called *exo-packets*.
- Exo-packets stored in packet buffer.
- Performs arithmetic and logical operations.
- Load and store *only* to its memory bank.

Execution slow compared to CPU . . .

. . . but can quickly react to changes in memory.

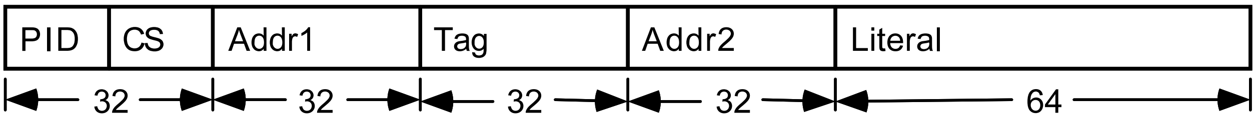
Exo-Packet

Exo-Packet Contents

- Procedure identifier. (PID)
- Control state (CS) ($\approx$ program counter).
- Literal operands.
- Memory-address operands.

  One memory address used for routing.

Example: Two-Operand Exo-Packet (Base System)



Execution Overview

CPU issues *exo-packet* . . .

. . . without blocking.

Exo-packet sent to memory unit holding the address.

Exo-processor typically:

Reads operand from memory . . .

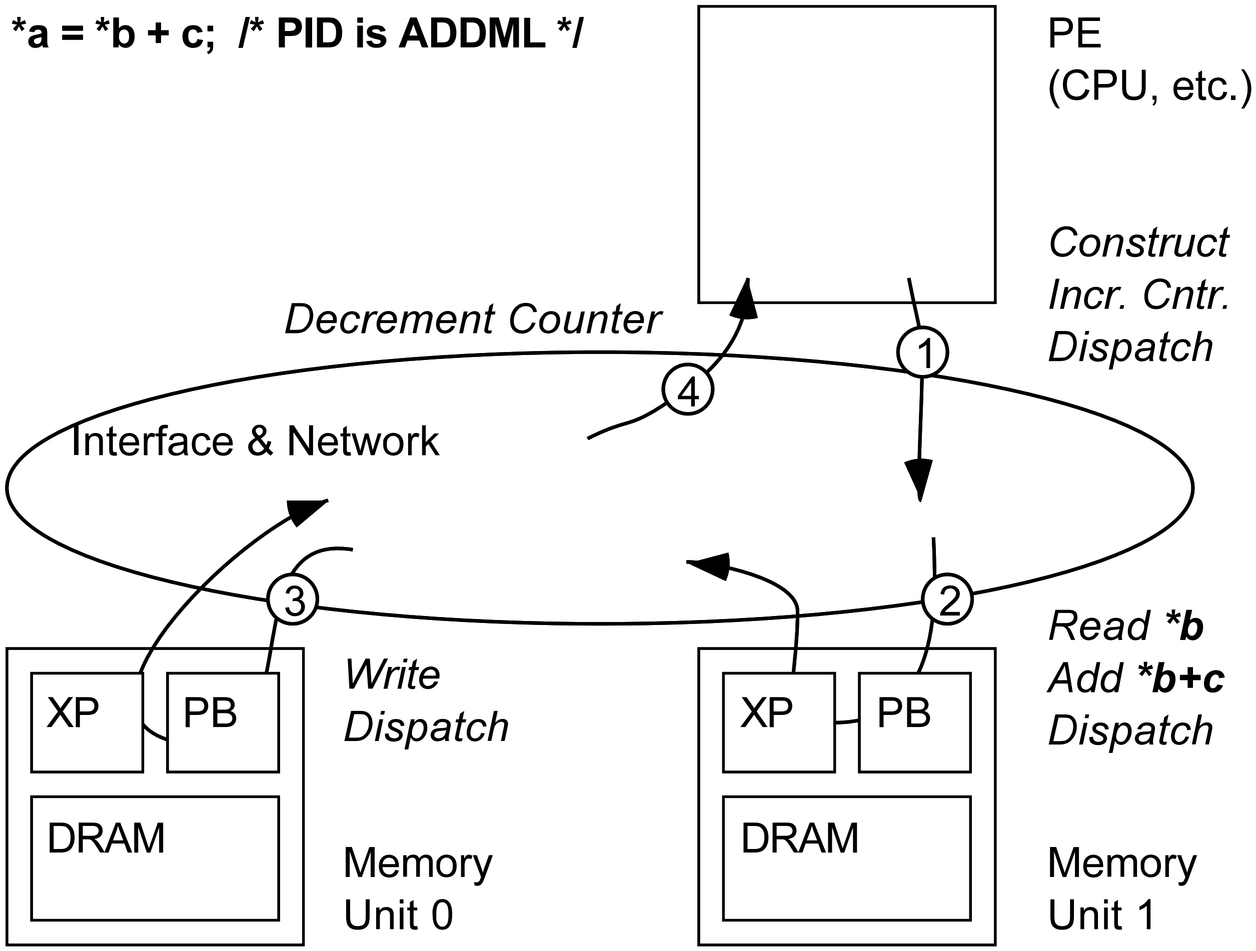
. . . performs operation . . .

. . . sends exo-packet to another exo-proc.

Operations that might be performed in a step:

- Read, operate, write to exo-packet.
- Read, operate, write to same address.
- Read, wait, operate, write tag, etc.

Exo-Op `*a = *b + c` Example



1: Dispatched Packet

ADDML	Step 1	`c`	`a`	`b`

2: Dispatched Packet

ADDML	Step 2	`*b+c`	`a`

3: Dispatched Packet

ADDML	Step 3

4: Acknowledge: In-progress counter decremented.

Tagged Operations

Tag Overview

Memory locations are associated with *tags* . . .

. . . possibly not part of addressed storage.

Memory operations test and set tags.

Similar to presence bits, full/empty bits, etc. . . .

. . . except that may be integers (requiring storage).

Typical Operation

1: Check tag against needed value.

2: If not equal, wait.

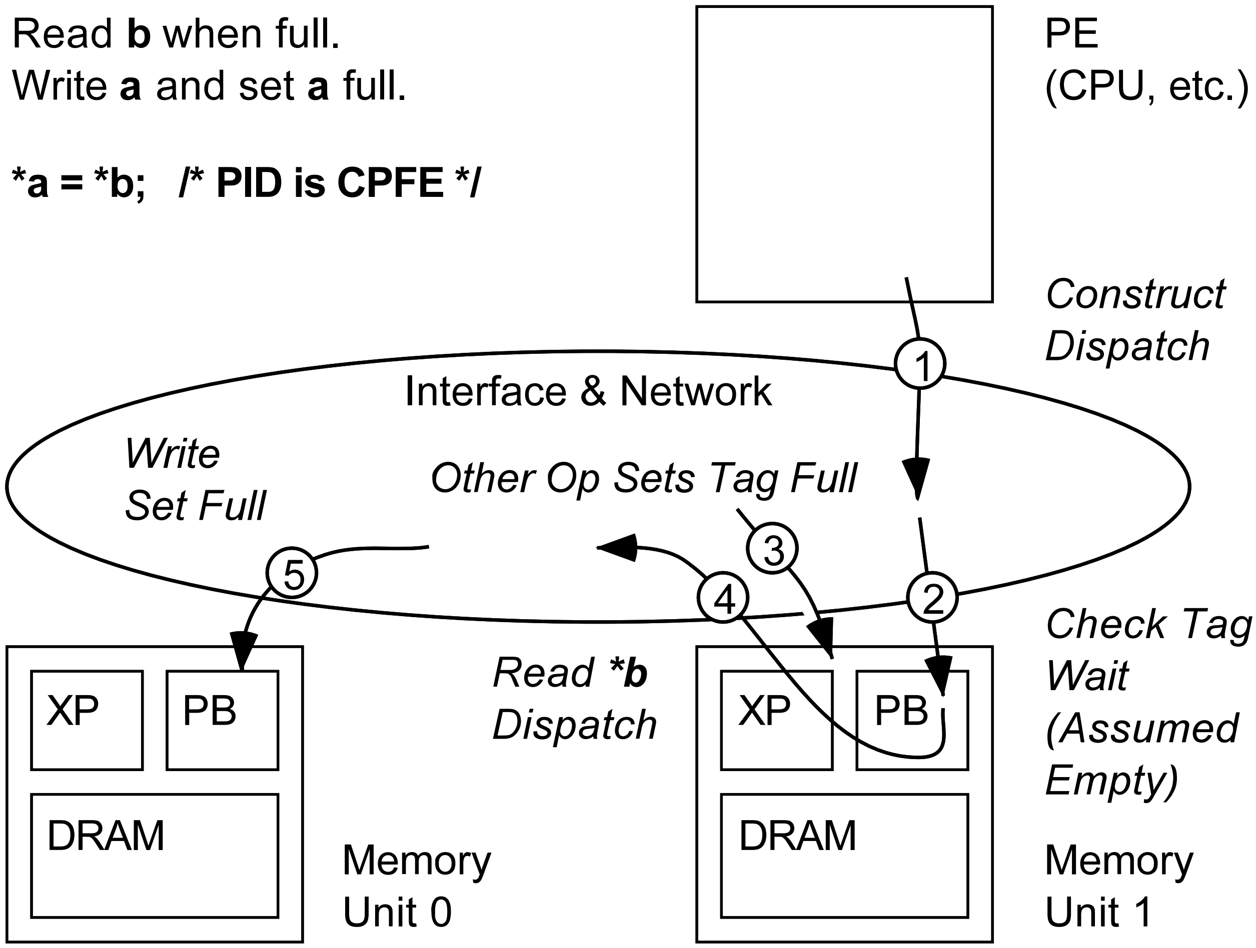
3: When equal, complete operation . . .

. . . possibly changing tag.

Exo-Op Tagged `*a = *b` Example

Read **b** when full.
Write **a** and set **a** full.

***a = *b;   /* PID is CPFE */**



1: Dispatched Packet

CPFE	Step 1	`a`	`b`

2: Tag assumed empty.

3: Other exo-op sets tag full.

4: Dispatched Packet

CPFE	Step 2	`a`	`*b`

5: No acknowledgment for this case.

Example, Nested Loops

Properties

- Two-level nested loop.
- Sums depend upon previous outer iteration.
- Sums within an outer iteration independent.
- One operand not statically determinable.

```
for( outer = 0; outer < outer_end; outer++ ){
  int *perm = perms[outer];

  /*  On odd iterations... */
  if( outer & 0x1 ) for( inner = 0; inner < size; inner++ )

       A[inner] = B[inner] + B[ perm[inner] ];

  /*  On even iterations... */
  else              for( inner = 0; inner < size; inner++ )

       B[inner] = A[inner] + A[ perm[inner] ];
}
```

Conventional Parallel Implementation

Changes

- Inner loop iterations partitioned evenly.
- Element A[i] and B[i] in nearby module ...
  ... for start $\leq i <$ stop.

  Thus, access fast for ≥ 2 of 3 accesses.

```
for( outer = 0; outer < outer_end; outer++ ){
  int *perm = perms[outer];  /* Return a permutation. */

  barrier();

  /*  On odd iterations... */
  if( outer & 0x1 ) for( inner = start; inner < stop; inner++ )

       A[inner] = B[inner] + B[ perm[inner] ];

  /*  On even iterations... */
  else for( inner = start; inner < stop; inner++ )

       B[inner] = A[inner] + A[ perm[inner] ];
}
```

Conventional Parallel Implementation

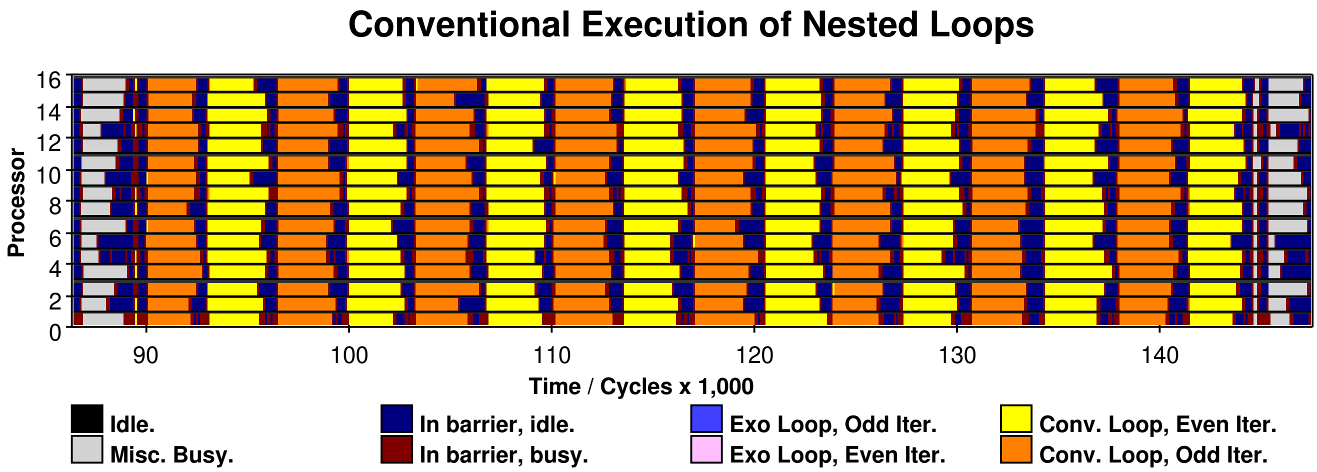
Execution Delays

- Possible miss for non-local access.
- Barrier overhead.
- Conservative control dependencies due to barrier.

```
for( outer = 0; outer < outer_end; outer++ ){
  int *perm = perms[outer];  /* Return a permutation. */

  barrier();

  /*  On odd iterations... */
  if( outer & 0x1 ) for( inner = start; inner < stop; inner++ )

       A[inner] = B[inner] + B[ perm[inner] ];

  /*  On even iterations... */
  else for( inner = start; inner < stop; inner++ )

       B[inner] = A[inner] + A[ perm[inner] ];
}
```

Execution of Conventional Loop Code

9-3



Gaps indicate barrier delays.

(Over) Simplified Exo Implementation

Changes

- Separate storage for adjacent iterations.
- Write sets tag FULL (not shown).
- Read waits for FULL tag.

```
for( outer = 0; outer < outer_end; outer++ ){

  int *perm = perms[outer];  /* Return a permutation. */

  for( inner = start; inner < stop; inner++ )

    A[outer+1][inner] =
       A[outer][inner]          /; FULL
     + A[outer][ perm[inner] ]  /; FULL ;

}}
```

Compilation

Add procedure compiled for exo-proc.

Execution at CPU

Exo-packet dispatched for each inner iteration.

No blocking: CPU does not wait after dispatch.

```
for( outer = 0; outer < outer_end; outer++ ){

  int *perm = perms[outer];  /* Return a permutation. */

  for( inner = start; inner < stop; inner++ )

    A[outer+1][inner] =
       A[outer][inner]           /; FULL
     + A[outer][ perm[inner] ]  /; FULL ;

}}
```

Execution Pacing

No dependencies will stall loop . . .

. . . but may stall to avoid congestion.

Actual (Simulated) Exo Implementation

9-6

If `outer` is large, substantial storage needed.

Solution 1:

Write only after two reads finish.

Solution 1 Disadvantages

Elements *must* be accessed twice.

A lot of packet buffer storage used.

Solution 2 (used here):

Provide storage for x outer iterations.

Assume one PE rarely $> x$ ahead of others.

Storage is reused, tag identifies outer iteration.

Reads check tag . . .

. . . if too low, they wait . . .

. . . if too high, fault routine called.

Solution 2 Disadvantages

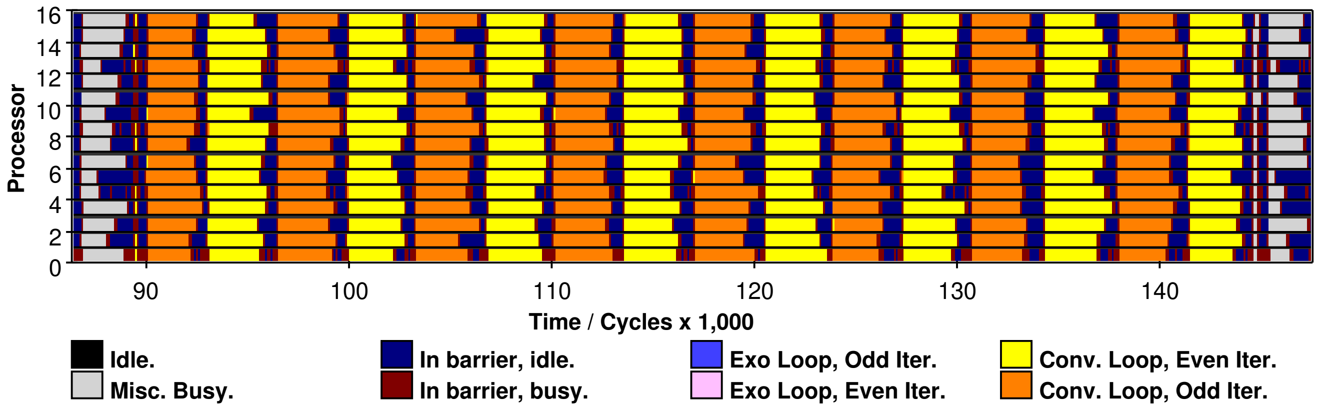
More storage than solution 1.

Re-starting after fault expensive or impossible.

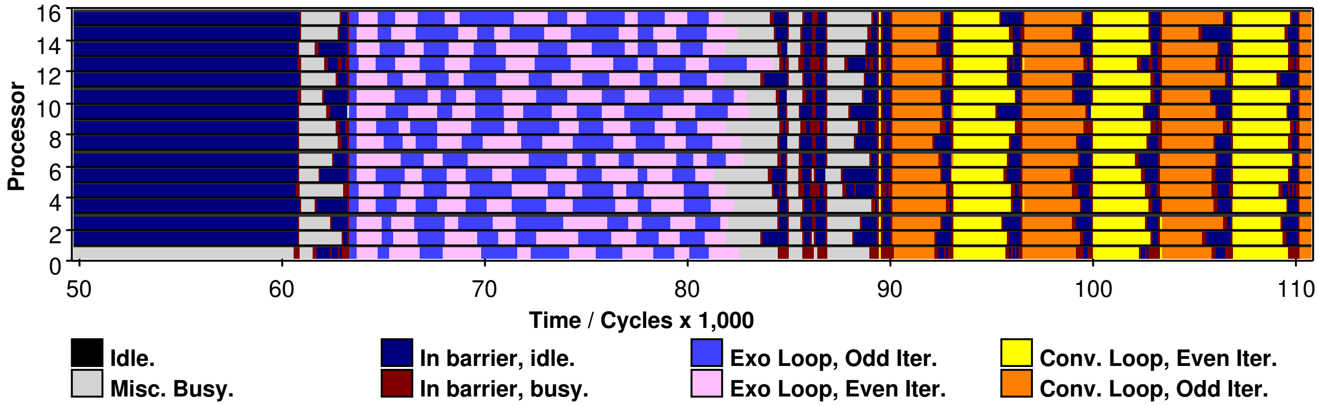
Execution Illustration

Execution

Conventional Execution of Nested Loops



Exo and Conventional (Partial) Execution of Nested Loops

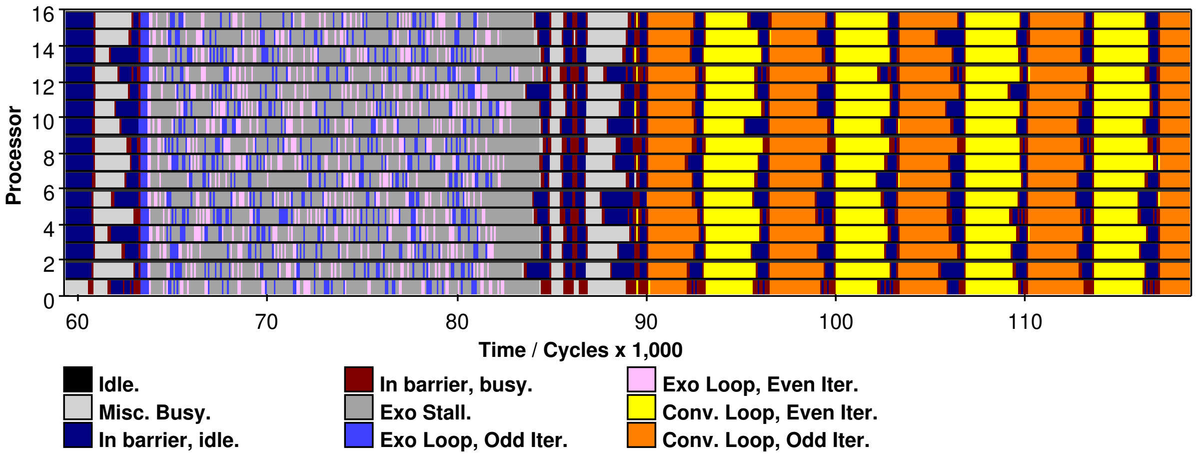


Note overlap in exo execution.

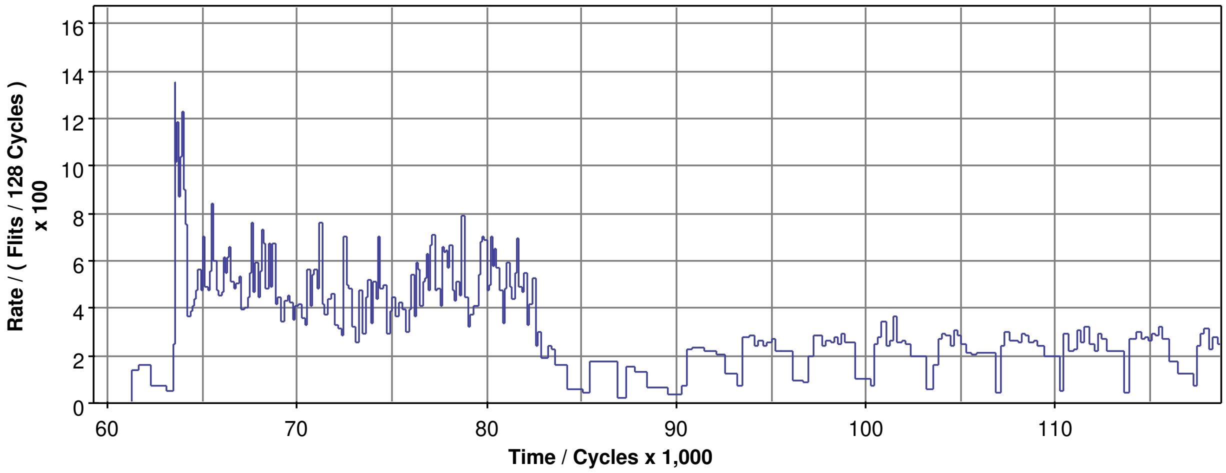
Note barrier gaps in conventional execution.

Execution Illustration

Stalls Due to Exceeded Threshold



Traffic Generated



Experimental Evaluation

Goal, determine sensitivity to:

- Network latency and bandwidth.
- Memory speed.
- Exo-op execution time.

Goal, determine speedup of:

- Program fragments with selected properties.
- A radix sort program.

Methodology

Execution-driven simulation using Proteus L3.10.

Run illustrative code fragments . . .

. . . and modified SPLASH-2 radix kernel.

Vary performance of key elements ($e.g.$, link width).

Determine bottlenecks.

Programs

Histogram

Small operations on widely shared data.

Suffers from contention and false sharing.

Very well suited to exo-ops.

Nested Loops

Three-operand operations with dependencies.

No spatial locality on one source operand.

Well suited to exo-ops.

Vector Sum `(a[i]=b[i]+c)`

Spatial locality on all operands.

For conventional version, data not initially cached.

Not appropriate for exo-ops (as described above).

In-Place and Copy Permutation

No spatial locality on destination operand.

Conventional implementation uses temp. storage.

Favorable layout for conventional version.

Programs

SPLASH-2 Radix

Exo-ops Used For

- Prefix sum.
- Permutation.

Conventional implementation tuned.

Simulation Parameters (Base System)

System and Network

16 Processor, 4×4 Mesh

Network interface shared by CPU and memory modules.

3-Cycle Wire Delay

3-Byte Link Width

Exo-Ops

21-Cycle Exo-Op Step Latency

12-Cycle Exo-Op Step Resource Usage ($\approx$ initiation interval)

25 In-Progress Exo-Op Stall Threshold

Memory System

42-Cycle Memory Latency

Four Banks per Memory Module

8-Way, 16-Byte Line, 2^{11} Set, Set Associative Cache

3-Cycle Cache Hit Latency

Time Units

Time given in *scaled cycles*.

3 Scaled Cycles = 1 “Real” Cycle

Memory and Cache

32-Bit Address Space

Sequential Consistency Model (w/o Exo-Ops)

Full-Map Directory Cache Coherence

Four Banks per Memory Module

Memory Banks Buffered (last block accessed).

42-Cycle Memory “Miss” Latency

12-Cycle Memory “Hit” Latency

8-Way, 16-Byte Line, 2^{11} Set, Set Associative Cache

3-Cycle Cache Hit Latency

Exo-Packet Size

8 Bytes . . .

. . . plus data . . .

. . . plus four bytes per value tag.

Exo-Op Timing

Issue (CPU instruction slots)

2 Cycles + 1 Cycle per operand.

Construction Time (issue to network injection)

6 Cycles

Execution of Exo-Op Step

21-Cycle Latency

12-Cycle Resource Usage ($\approx$ initiation interval)

Plus, time for operand memory access.

Flow Control

25 In-Progress Exo-Op Stall Threshold

Simulation Results

For Base System

Program	Conv.	Exo	Speedup
Radix	13.5	9.5	1.4
Histo.	156.3	17.1	9.1
Loops	215.6	84.8	2.5
Vector	75.8	38.8	2.0
In Place Perm.	331.8	78.2	4.2
Copy Perm.	107.0	35.7	3.0

Timing given in cycles over input size.

Conclusions

- Whole-program speedup usable.
- Very significant speedup on portions.

Base System Bottlenecks

Can the rest of the system keep up with exo-op issue?

Of course not!

Consider a two-step, two-tag exo-op.

Issue time is 7 cycles.

If CPU repeatedly issued these, to keep up . . .

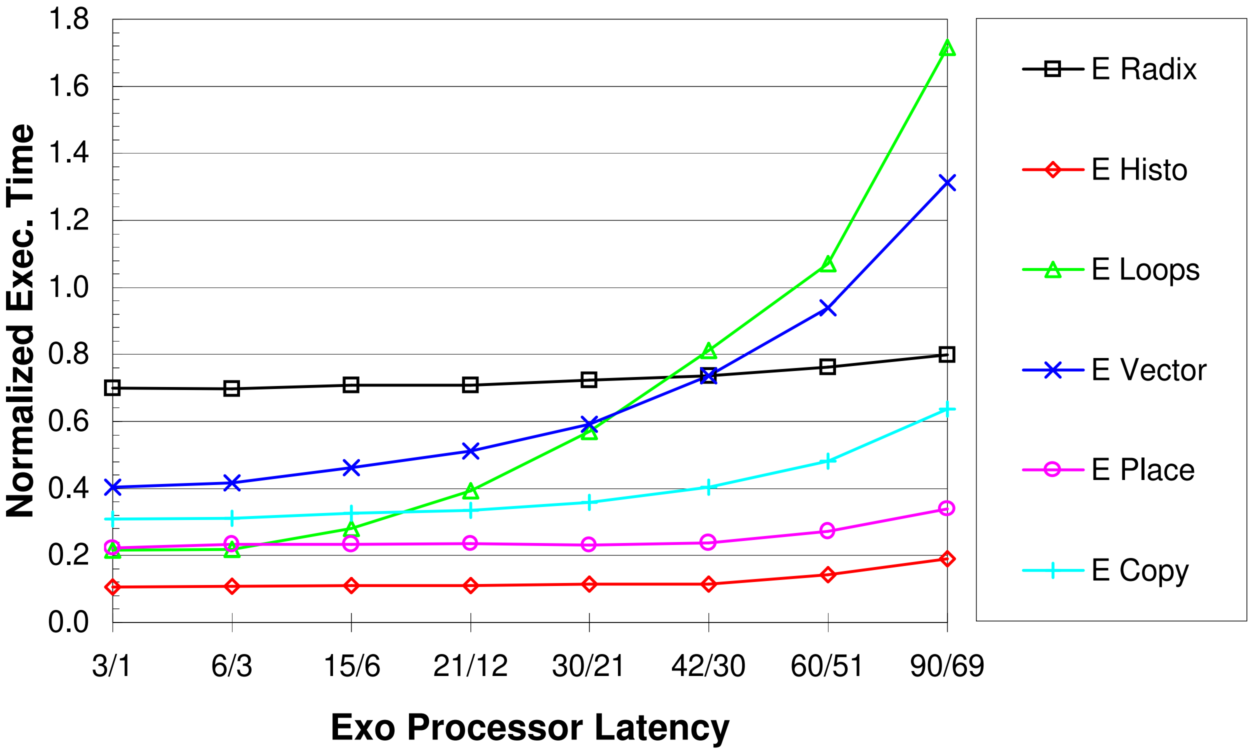
. . . memory would have to be $6\times$ faster . . .

. . . network interface would have to be $2.86\times$ faster . . .

. . . and exo-processor would have to be $1.71\times$ faster . . .

Not as unbalanced as numbers suggest.

Exo-Processor Speed Experiments



Exo-Processor Normally Faster Than Memory & Network

Sensitive Programs

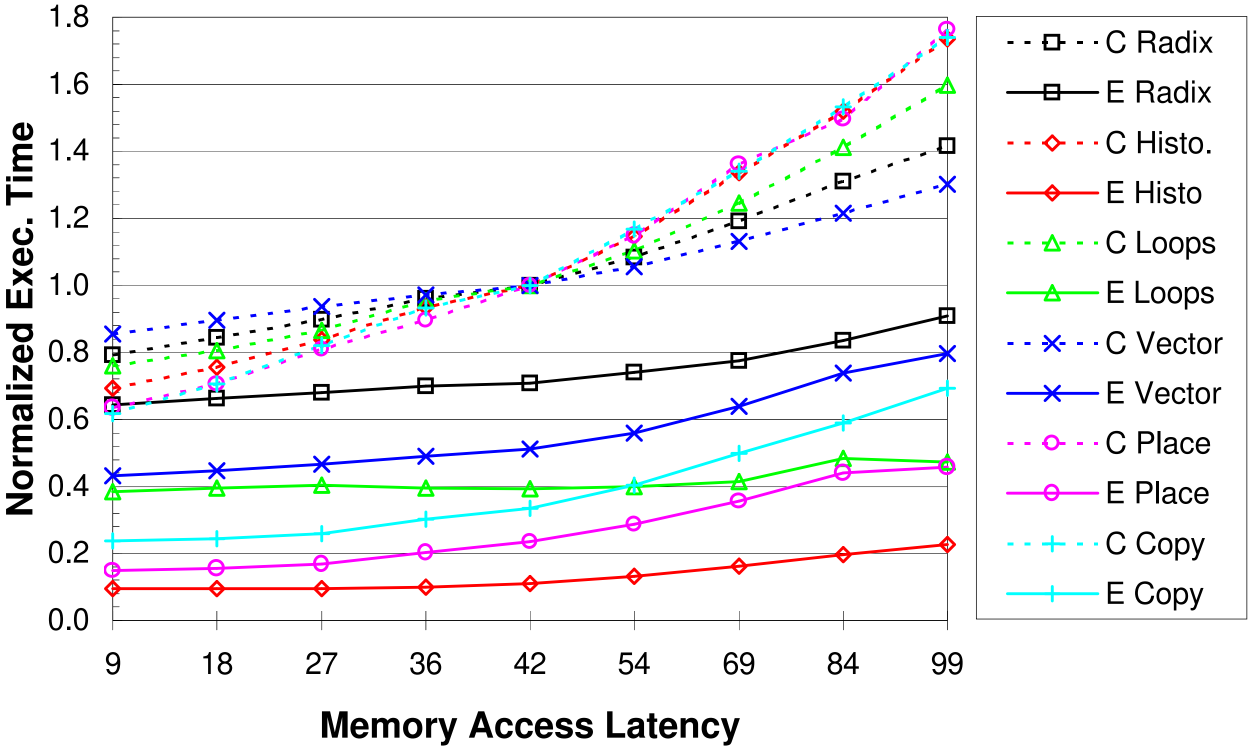
Nested loops, because of tags.

Vector, locality avoids memory bottleneck . . .

. . . exposing exo-processor bottleneck.

Others work well with slower exo-processor.

Memory Latency Experiments



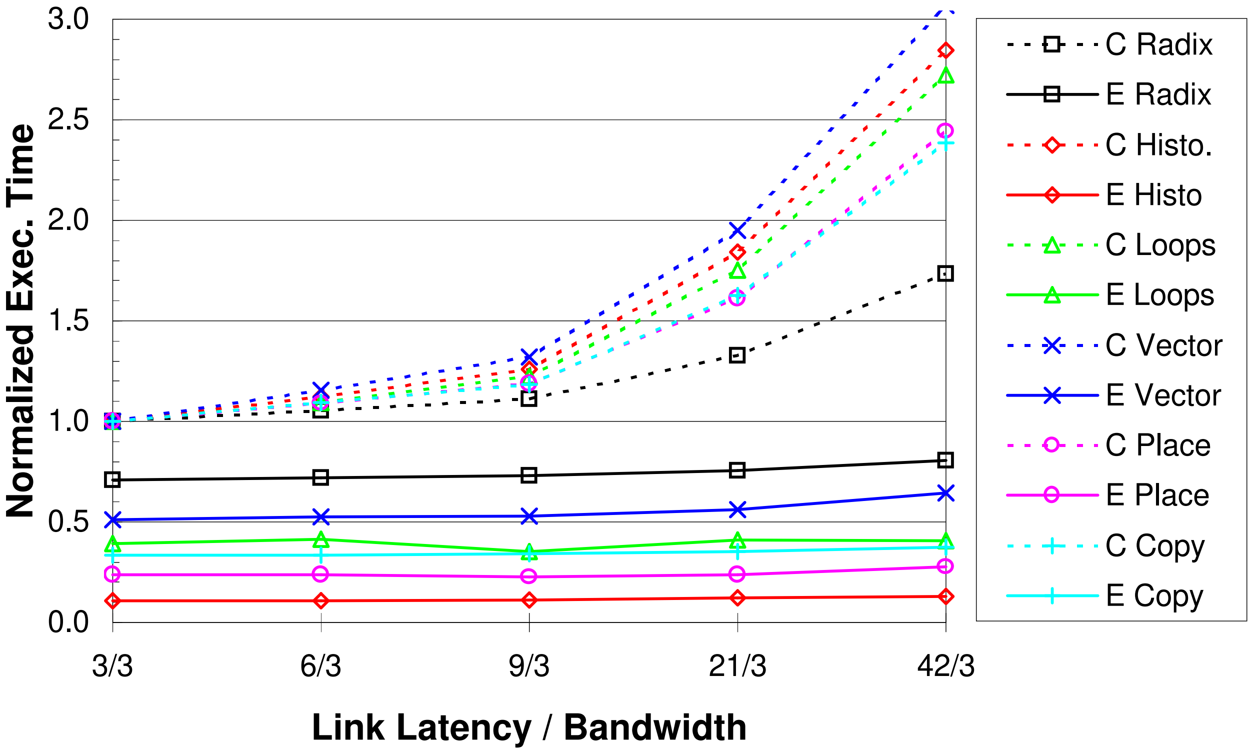
Conventional programs more affected.

Exo programs hide lower memory latency.

Loops less sensitive because of heavy exo-processor use.

Network Latency Experiments

18-2



x/y Notation:

x-cycle link delay (pipelined).

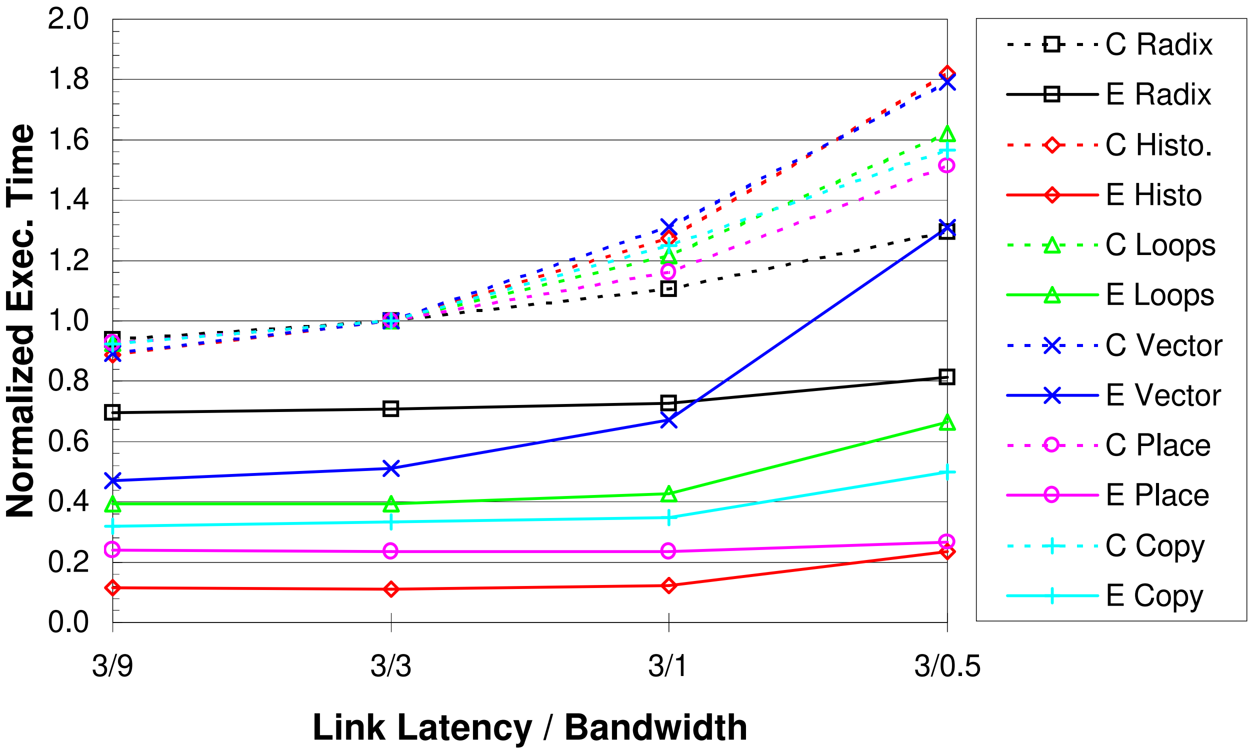
y-byte per cycle link bandwidth.

High Latency

Conventional Programs Suffer

Exo Programs Almost Unaffected

Network Bandwidth Experiments



x/y Notation:

x-cycle link delay (pipelined).

y-byte per cycle link bandwidth.

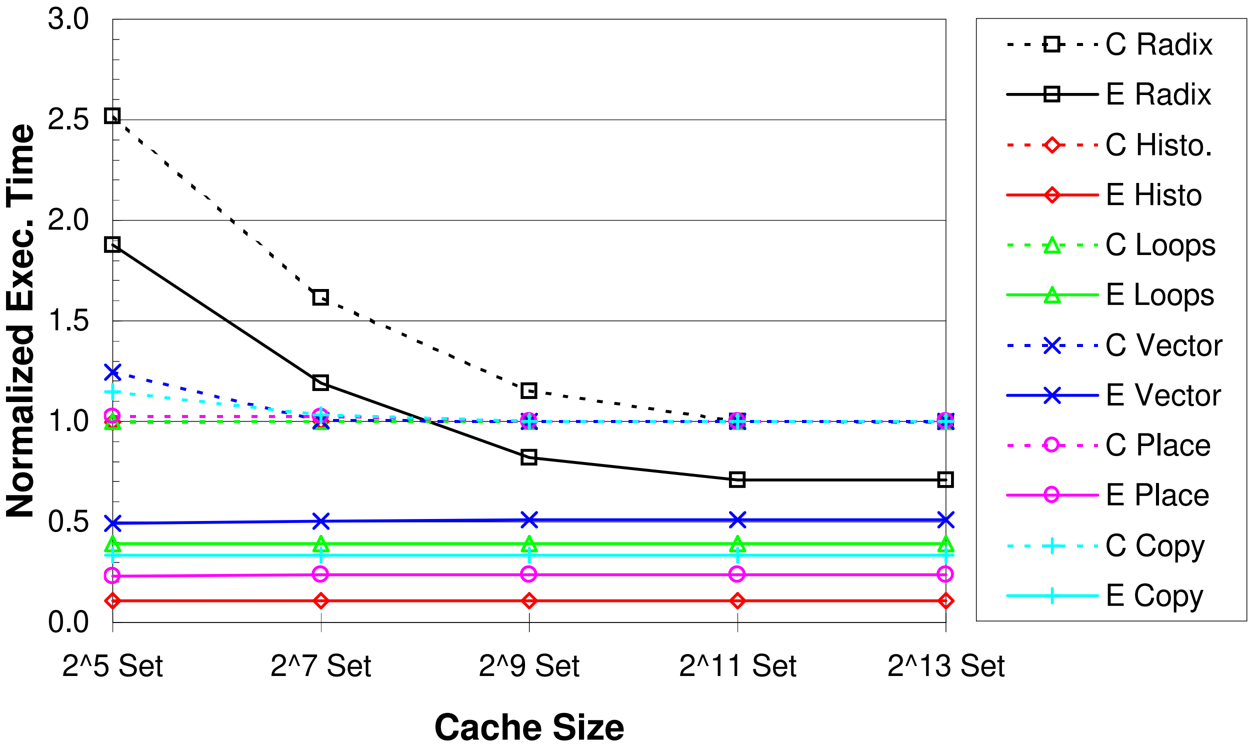
Low Bandwidth

Both exo and conventional programs slowed

Speedup nevertheless higher.

Cache Size Experiments

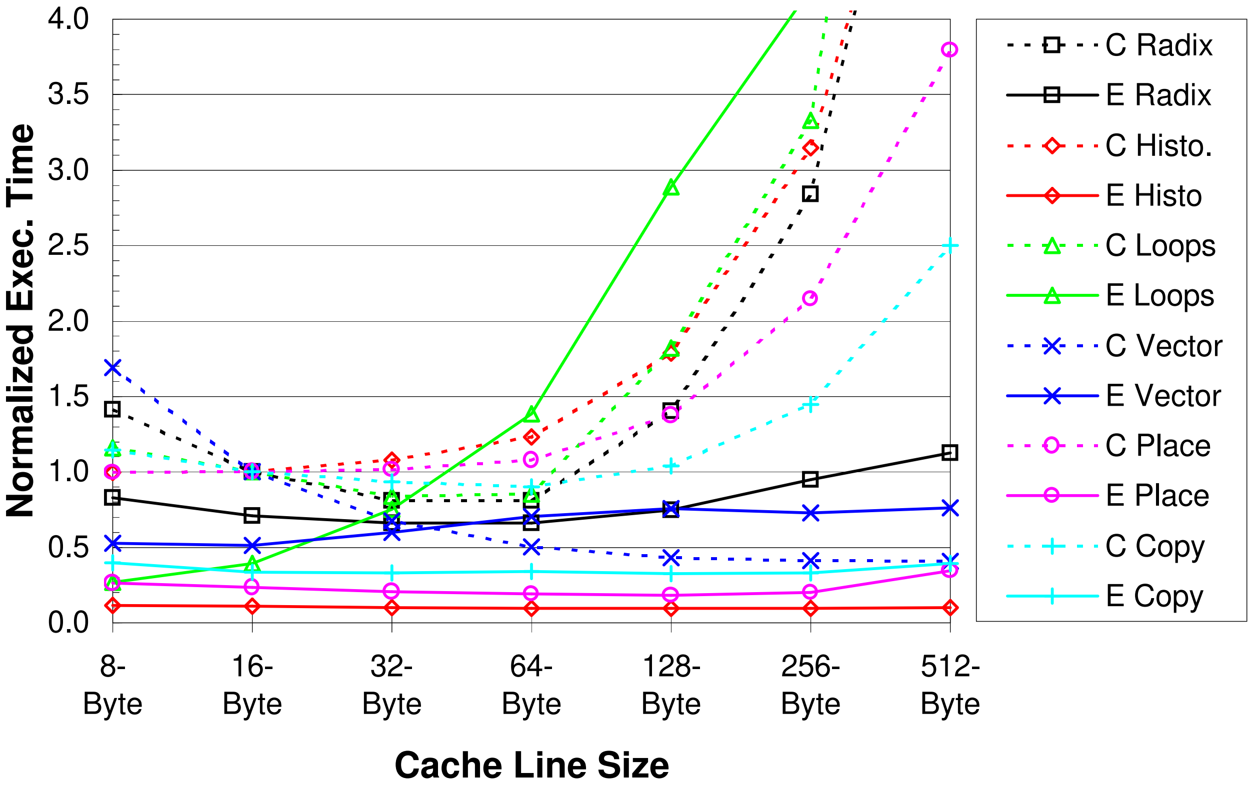
18-4



Fragments use a small amount of memory.

Small cache reduces speedup on radix.

Line Size Experiments

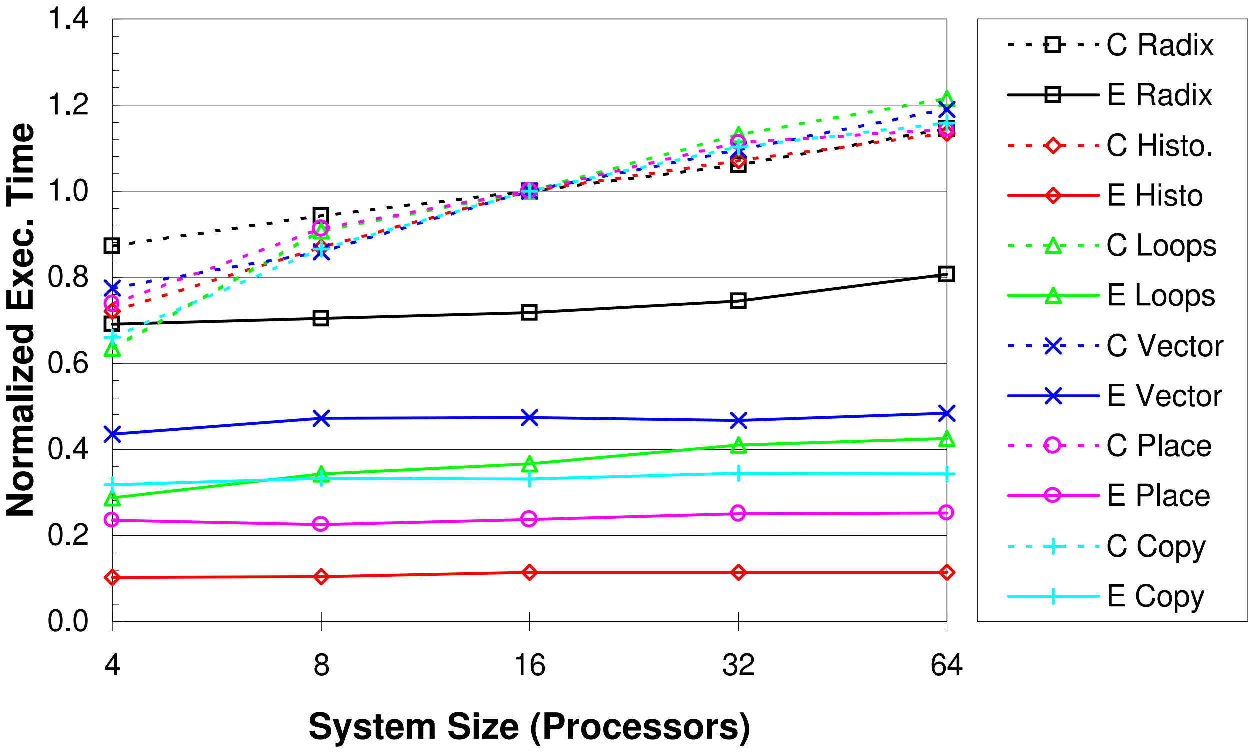


Tag wait mechanism works poorly with long lines.

Effect varies with access characteristics.

System Size (Processors)

18-6

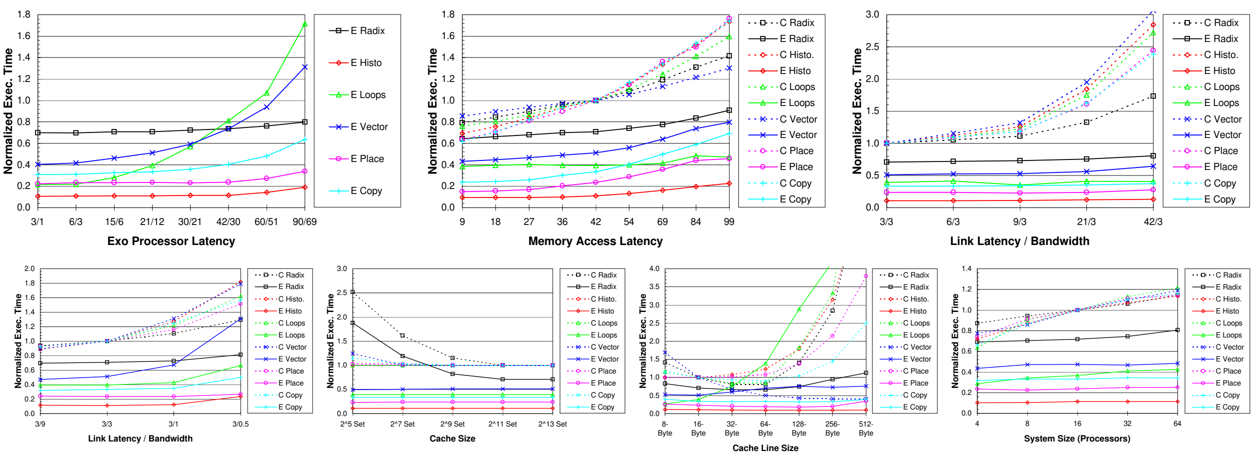


Input scaled (constant input per proc.) .

Because of latency hiding, exo better in large systems.

19

Results Summary



Exo-Processor Speed

Bottleneck in some programs.

Slower speed tolerable for others.

Memory Latency

Effectively hidden, slower memories usable.

Network Latency

Effectively hidden.

Network Bandwidth

Slows conventional and exo programs.

But speedup higher with lower bandwidth.

Related Work—Remote Operations

Nonlocally Executed Atomic Operations

E.g., fetch and add, compare and swap.

Implemented in Cray T3E, Cedar.

Similarities

CPU quickly issues operation.

Operations are small.

Difference

Because of size, limited applicability to computation.

Related Work—Remote Operations

Protocol Processors

Dedicated processor for coherence protocol.

Used in Stanford FLASH.

Similarity

Dedicated processor for memory.

Difference

No specific mechanism for tag waiting.

Not used for computation.

Related Work—Remote Operations

Active Message Machines

Mechanism provided for fast sending, receiving, and return.

Examples include $\star$T, EM-X, and the M-Machine

Similarities

Low-latency message dispatch and receipt.

Messages used for memory access.

Tag bits.

Differences

Message handlers may run on CPU slowing other tasks.

Employ multithreaded or dataflow paradigms ...

... which may favor different problems than exo systems.

Related Work—Latency Hiding

Low-Switch-Latency Multithreaded Parallel Machines

Some switch between threads in a few cycles.

Some can issue any active resident thread.

Used in many current research projects.

Similarity

Operation in one thread can start while another waits.

Difference

Overhead consumed for each thread ($e.g.$, index bounds).

Feasibility Assumptions

Programs exhibit misses, contention, and synch. delays.

- No well-behaved alternate implementations.
- Delays significantly impact execution time.
- There exist many such programs . . .
- . . . or users willing to pay for maximum performance.

Multithreading sometimes adds too much overhead.

Communication bandwidth not too expensive.

- Latency can be high.

Cost of exo-processor much less than CPU.

- An exo-processor *doesn't* need . . .
- . . . pipelined execution units . . .
- . . . cache, address translation, etc. . . .
- . . . or an additional network port.

Further Work

Algorithm-Driven Development

Implement algorithms too fine-grained for current machines.

Vector Exo-Ops

Exploit spatial locality using large word sizes in DRAM.

Comparison with Multithreaded Machines

Compare exo-scalar and multiple-thread overheads.

Compare ease of discovering multiple threads and exo-ops.

Comparison with Cache-Visible Architectures

Programs can bypass cache.

Subblock writes at memories.

Etc.