Benchmark-driven Approaches to Performance Modeling of Multi-Core Architectures

Ph.D Defense

Bertrand Putigny

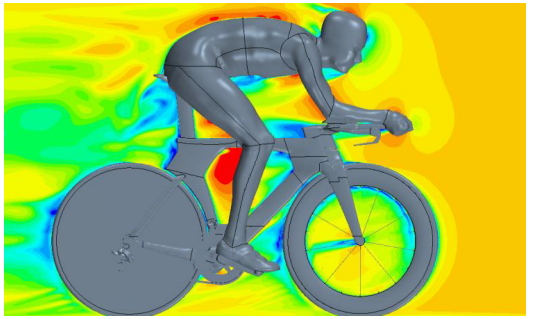
March 27, 2014

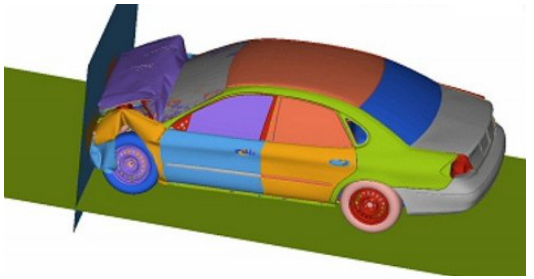
Outline

Context

Numerical simulation

- Engineering
  - Aerodynamics
  - Crash simulation
- Weather forecast
- Gas/oil exploration
- Quantum mechanics





Purpose

- Reducing costs
- Model testing
- Experimental science: *e.g.,* astrophysics simulation

High Performance Computing

Computer simulation

- Requires great computing resources
- Larger model
- Reducing computation time

$\Rightarrow$ growing need for resources

How to sustain the growing need for computing resources

- Faster processors (frequency)
- More features
  - vector instructions
  - out-of-order
- Parallel processors
- Clusters

High Performance Computing

Computer simulation

- Requires great computing resources
- Larger model
- Reducing computation time

$\Rightarrow$ growing need for resources

How to sustain the growing need for computing resources

- Faster processors (frequency)
- More features
  - vector instructions
  - out-of-order
- Parallel processors
- Clusters

High Performance Computing

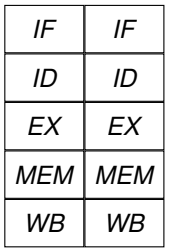
Computer simulation

- Requires great computing resources
- Larger model
- Reducing computation time

$\Rightarrow$ growing need for resources

How to sustain the growing need for computing resources

- Faster processors (frequency)
- More features
  - vector instructions
  - out-of-order
- Parallel processors
- Clusters



High Performance Computing

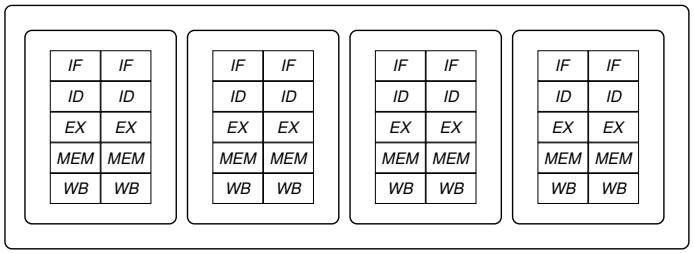
Computer simulation

- Requires great computing resources
- Larger model
- Reducing computation time

$\Rightarrow$ growing need for resources

How to sustain the growing need for computing resources

- Faster processors (frequency)
- More features
  - vector instructions
  - out-of-order
- Parallel processors
- Clusters



High Performance Computing

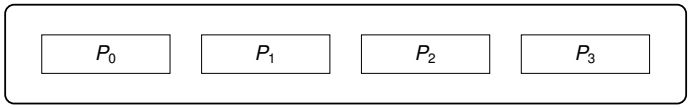
Computer simulation

- Requires great computing resources
- Larger model
- Reducing computation time

$\Rightarrow$ growing need for resources

How to sustain the growing need for computing resources

- Faster processors (frequency)
- More features
  - vector instructions
  - out-of-order
- Parallel processors
- Clusters



High Performance Computing

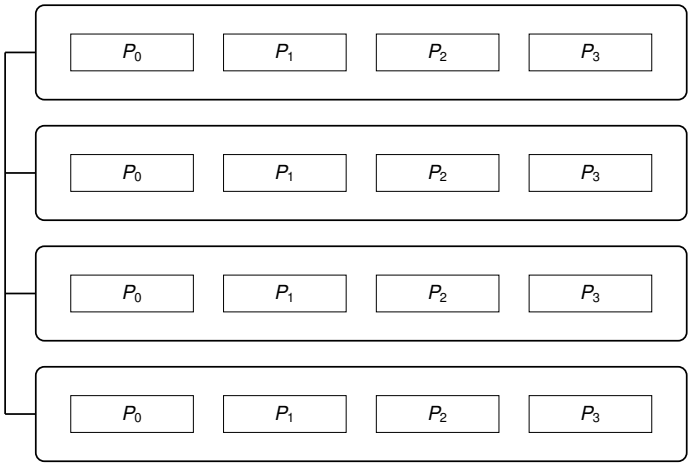
Computer simulation

- Requires great computing resources
- Larger model
- Reducing computation time

$\Rightarrow$ growing need for resources

How to sustain the growing need for computing resources

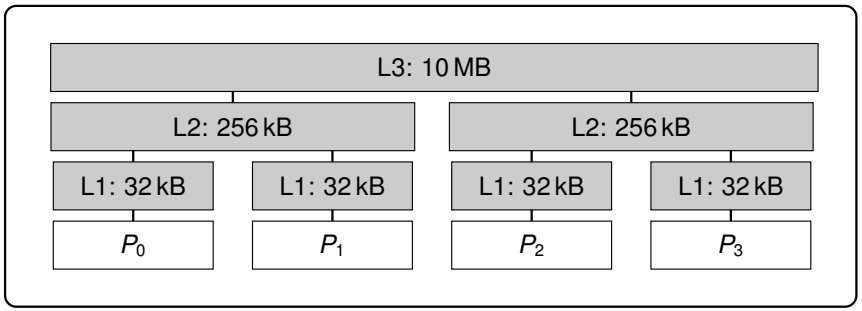
- Faster processors (frequency)
- More features
  - vector instructions
  - out-of-order
- Parallel processors
- Clusters



Memory Hierarchy and Cache Coherence

Caches

- Small and fast memories
- Hardware managed



Cache coherence

- Caches: data replication
- Maintain memory consistency

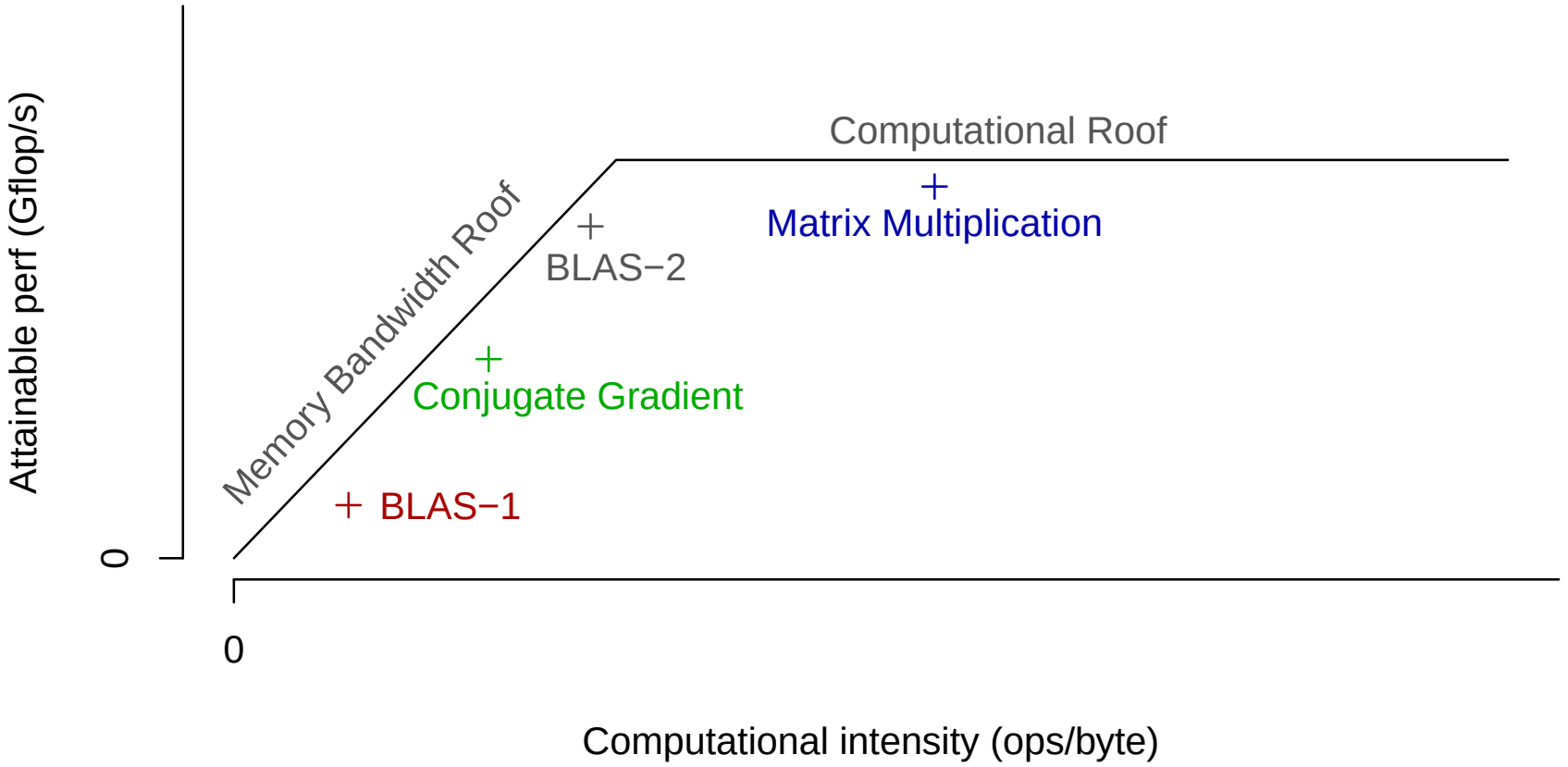
Dealing with Hardware Complexity

Performance modeling

- Simplified view of the hardware
- Performance insight
  - Compilers
  - Tools (MAQAO, CacheGrind, TAU, Scalasca . . .)
  - Users
- New hardware: automatic or portable

Performance Models

- Roofline model [Patterson 2009]
- Stack model [Mattson 1970]
- Higher level: bulk-synchronous model [Culler 1993], LogP [Valiant 1990],



Issues with current Performance Models

- Hardware documentation
  - Availability and accuracy questionable
    e.g., Sandy-Bridge: `divps` instruction latency:
    - Intel's doc: 39 cycles
    - Agner Fog: 10-14 cycles
- Irrelevant metrics usage (*e.g.,* number cache miss)
- Inaccurate prediction due to simplifications

Proposition

- Benchmarks to calibrate models
  - Bridge the gap between models and reality
- Aim for time prediction
  - Optimization objective function

Outline

Performance Modeling

Software (from the hardware point of view)

Instruction flow $\Rightarrow$ predicting execution time of each instruction

But it is not enough

- Super-scalar pipeline
- Out-of-order execution

Even worse

- Some instruction execution times depend on hardware state (*e.g.,* memory access)
- Hardware automatic feature: complicated understanding and prediction

Computational Model

Goal

Predicting a basic block execution time

```
MULPD    XMM0, XMM1
MULPD    XMM2, XMM3
ADDPD    XMM1, XMM2
MULPD    XMM3, XMM4
MULPD    XMM5, XMM6
ADDPD    XMM4, XMM5
```

We need

- Individual instruction execution time
- Number of parallel instructions
- Conflicting instructions

Motivation

- Basis for compiler optimizations
- Optimization and code transformation decision

Computational Model

Related work

- Benchmarking frameworks
  - LIKWID [Treibig 2010]
  - MicroTools [Beyler 2012]
- Hardware documentation
  - Often incomplete
- Agner Fog's instruction latencies (`x86`, AMD and VIA)
  - Hand-tuned benchmarks
  - No automatic measurement method

Contribution 1: Measuring Instruction Latency

During James Tombi's internship

Input example x86 architecture

Registers

```
R64: [EAX, EBX, ECX, EDX, ...]
XMM: [XMM0, XMM1, ..., XMM15]
YMM: [YMM0, YMM1, ..., YMM15]
```

Instruction syntax

```
ADD     R64, R64
MOV     R64, R64
ADDPD   XMM, XMM
MULPD   XMM, XMM
...
```

produces

- Several code versions
  - Loop unrolling
  - Register pressure
- Several code patterns
  - Latency: dependence chain
  - Throughput: no dependence

```
latency:                    throughput:
ADDPD   XMM0, XMM1          ADDPD   XMM0, XMM1
ADDPD   XMM1, XMM2          ADDPD   XMM2, XMM3
ADDPD   XMM2, XMM3          ADDPD   XMM4, XMM5
ADDPD   XMM3, XMM4          ADDPD   XMM6, XMM7
```

Measurement Results

CPU: Sandy-Bridge Xeon E5-2650 (2.00 GHz)

Timer: `RDTSC`

Unroll: 64

Instruction	Operand	Throughput		Latency	
		Model	Agner Fog	Model	Agner Fog
ADD	imm/r64, r64	0.34	0.33	1.00	1
MOV	imm/r64, r64	0.34	0.33	1.00	1
INSERTPS	imm, xmm, xmm	1.01	1	1.02	1
SHUFPS/D	imm, xmm, xmm	1.01	1	1.02	1
ADDPS/D	xmm, xmm	1.02	1	3.00	3
ANDPS/D	xmm, xmm	1.01	1	1.02	1
MOVAPS	xmm, xmm	1.01	1	1.02	1
ORPS/D	xmm, xmm	1.01	1	1.02	1
MAXPS/D	xmm, xmm	1.02	1	3.00	3
MINPS/D	xmm, xmm	1.02	1	3.00	3
HSUBPS/D	xmm, xmm	2.01	2	5.00	5
MOVQ	xmm, xmm	0.34	0.33	1.00	1
MOVSS/D	xmm, xmm	1.01	1	1.02	1
MULPS/D	xmm, xmm	1.02	1	5.00	5
PADDB/W/D/Q	xmm, xmm	0.51	0.5	1.01	1
PAND	xmm, xmm	0.34	0.33	1.00	1

Functional Unit Sharing

- Interleaving instructions
- For each couple of instructions
- Merge both instruction throughput codes

Example

`mov` instruction (throughput 0.33) and `fadd` (throughput 1):

```
MOV   R8D,  R9D;
MOV   R10D, R11D;
MOV   R12D, R13D;
MOV   R8D,  R9D;
MOV   R10D, R11D;
MOV   R12D, R13D;
```

time: 2 cycle

+

```
FADD ECX;
FADD ECX;
```

time: 2 cycle

$\Rightarrow$

```
MOV   R8D,  R9D;
MOV   R10D, R11D;
MOV   R12D, R13D;
FADD ECX;
MOV   R8D,  R9D;
MOV   R10D, R11D;
MOV   R12D, R13D;
FADD ECX;
```

2 cycle or more?

On-Core Modeling: Summary

Achievements

- Approach used to build MAQAO static model for the Xeon Phi
- Deeper understanding of processor performance
  - Where bottlenecks are
  - Code transformation impact
- Performance modeling of the SCC chip for power efficiency optimization[1]

Compared to related work

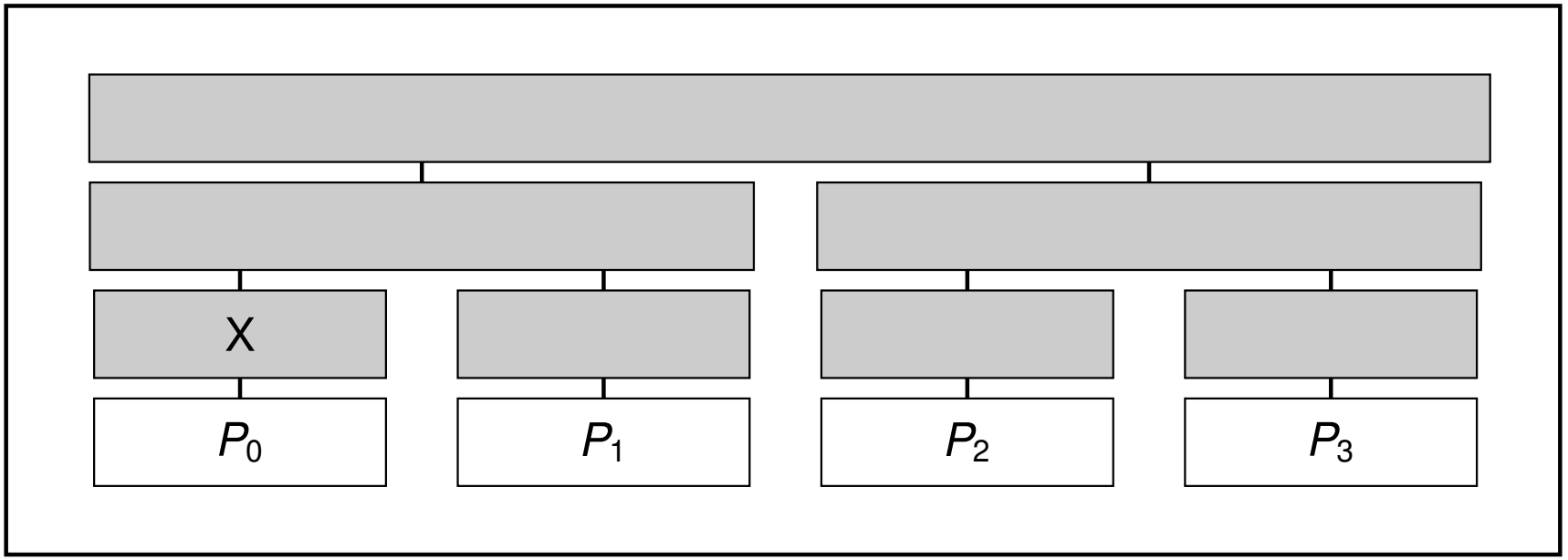
- Automatic instruction performance measurement
- Automatic execution port sharing detection

[1] B. Putigny, B. Goglin, D. Barthou. Performance modeling for power consumption reduction on SCC. In Proceedings of the 4th Many-core Applications Research Community (MARC) Symposium. 2012

Outline

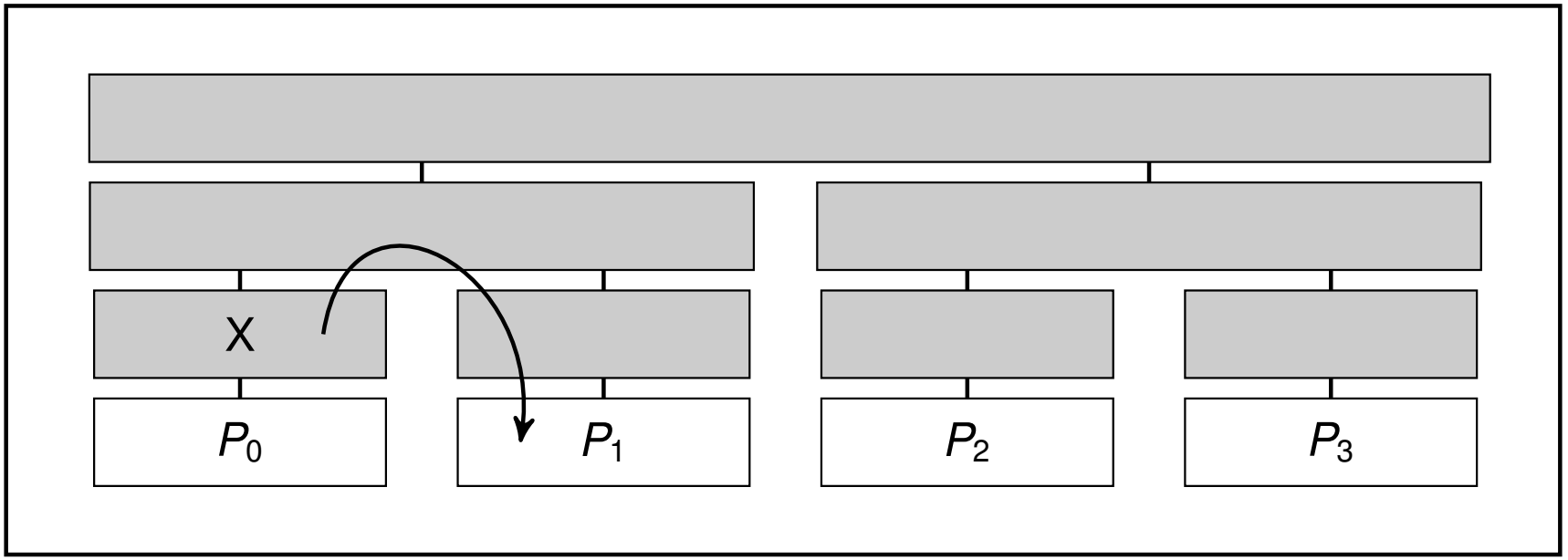
Cache Coherence Impact on Performance

Program 1	
Thread 0	Thread 1
Read X	
	Read X



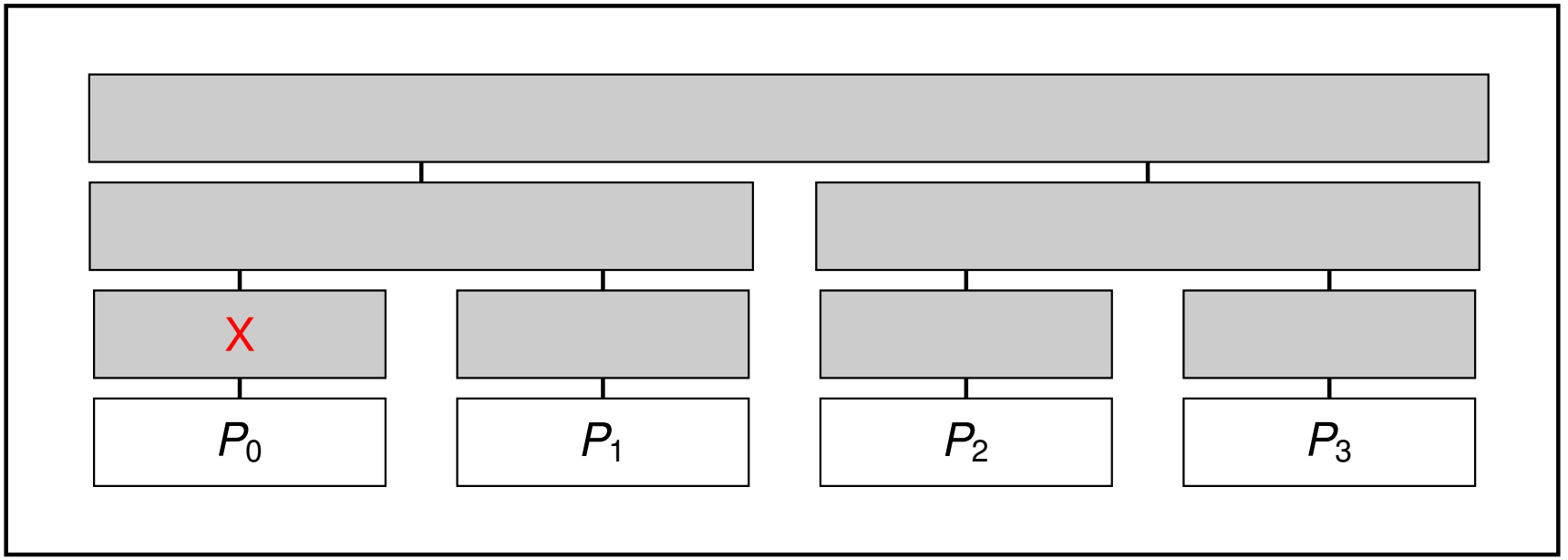
Cache Coherence Impact on Performance

Program 1	
Thread 0	Thread 1
Read X	
	Read X



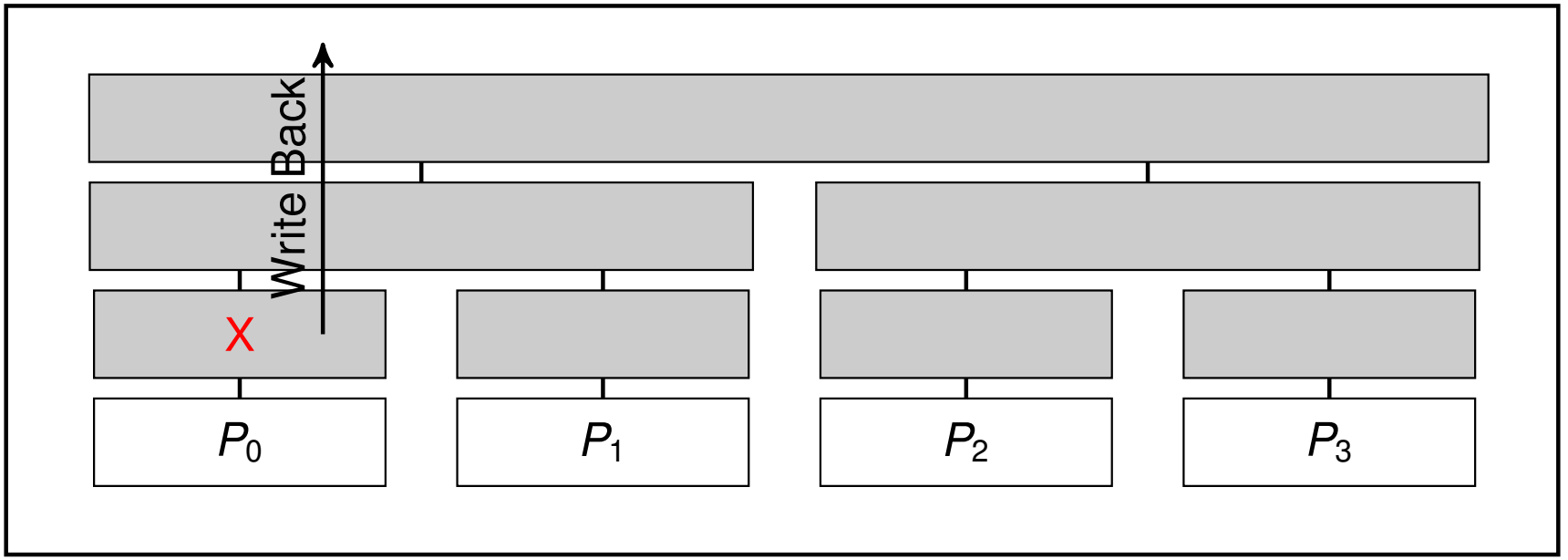
Cache Coherence Impact on Performance

Program 2	
Thread 0	Thread 1
Write X	
	Read X



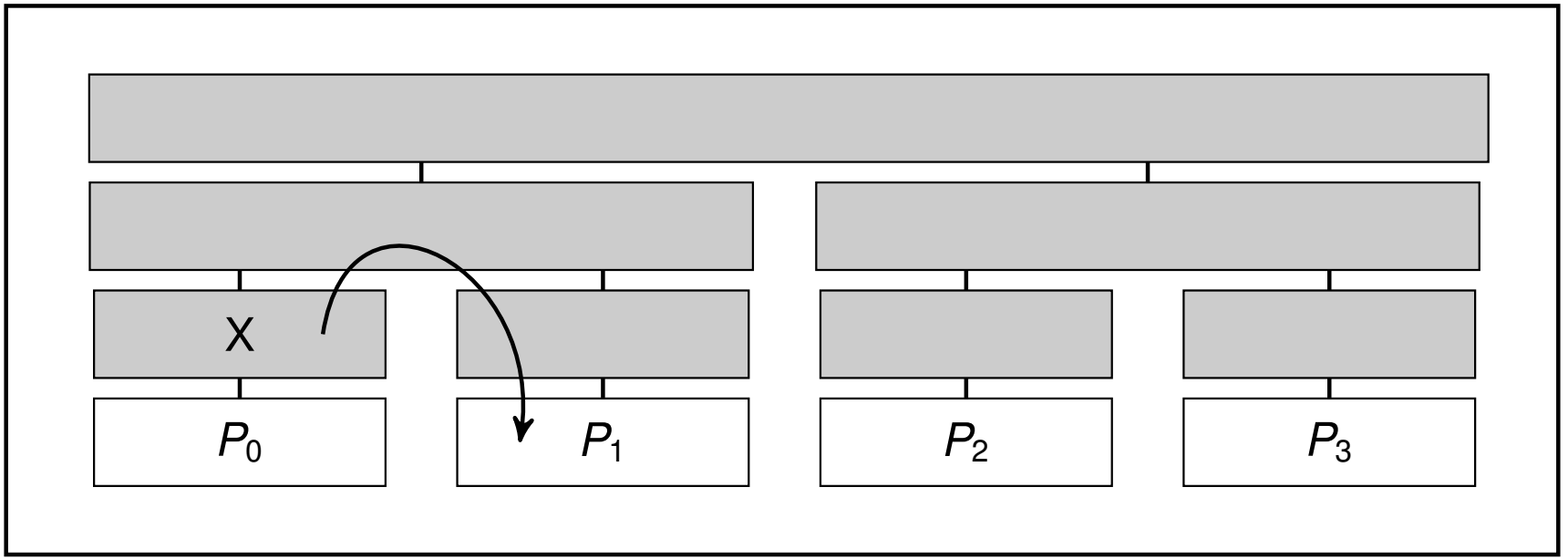
Cache Coherence Impact on Performance

Program 2	
Thread 0	Thread 1
Write X	
	Read X



Cache Coherence Impact on Performance

Program 2	
Thread 0	Thread 1
Write X	
	Read X

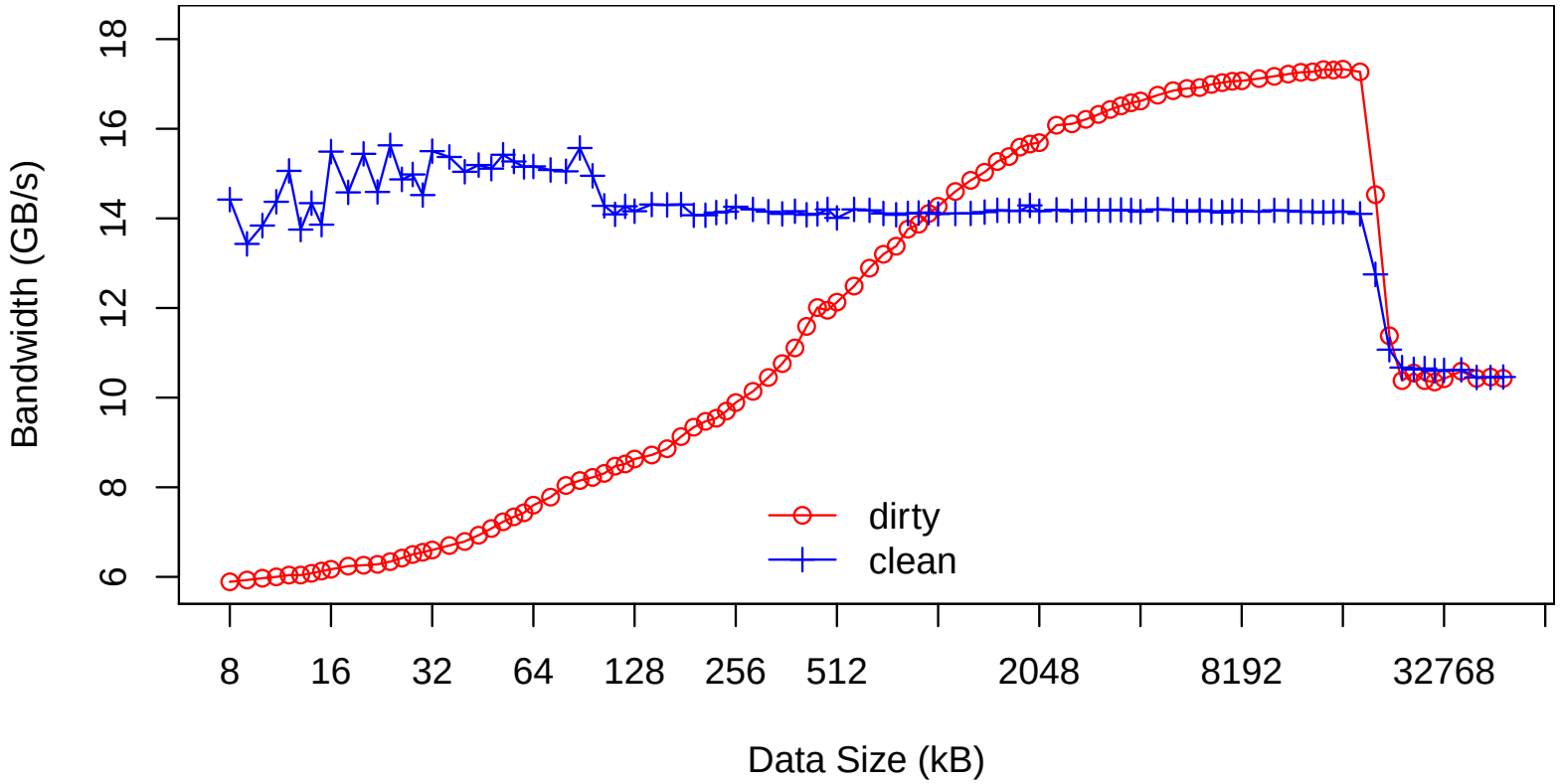


Cache Coherence Impact on Performance

Program 1	
Thread 0	Thread 1
Read X	
	Read X

Program 2	
Thread 0	Thread 1
Write X	
	Read X

Real experiment (Sandy-Bridge architecture):



Memory Models: the 5Cs

	stack model	simulators
Compulsory		
Capacity		
Contention		
Coherence		
Conflict		

Memory Models: the 5Cs

	stack model	simulators
Compulsory	✓	✓
Capacity		
Contention		
Coherence		
Conflict		

Memory Models: the 5Cs

	stack model	simulators
Compulsory	✓	✓
Capacity	✓	✓
Contention		
Coherence		
Conflict		

Memory Models: the 5Cs

	stack model	simulators
Compulsory	✓	✓
Capacity	✓	✓
Contention	✗	✓
Coherence		
Conflict		

Memory Models: the 5Cs

	stack model	simulators
Compulsory	✓	✓
Capacity	✓	✓
Contention	✗	✓
Coherence	✗	✓
Conflict		

Memory Models: the 5Cs

	stack model	simulators
Compulsory	✓	✓
Capacity	✓	✓
Contention	✗	✓
Coherence	✗	✓
Conflict	✗	✓

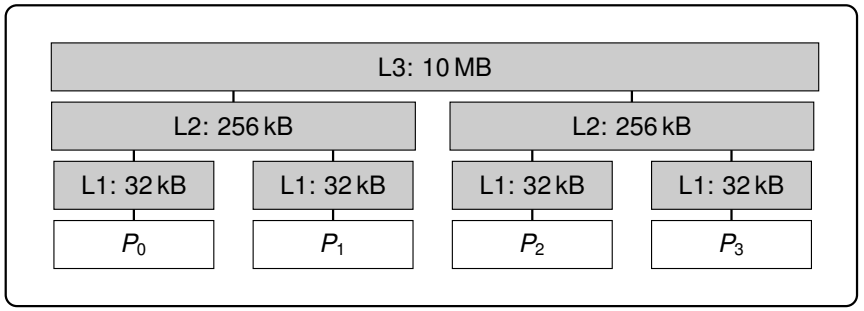
Proposed Memory Model: Overview

Model decomposed into:

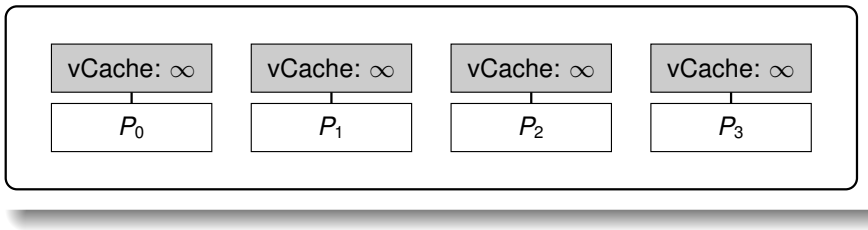
1. Hardware Model
2. Software representation
3. Chunk state tracking
4. Time prediction

Step 1: Hardware Model

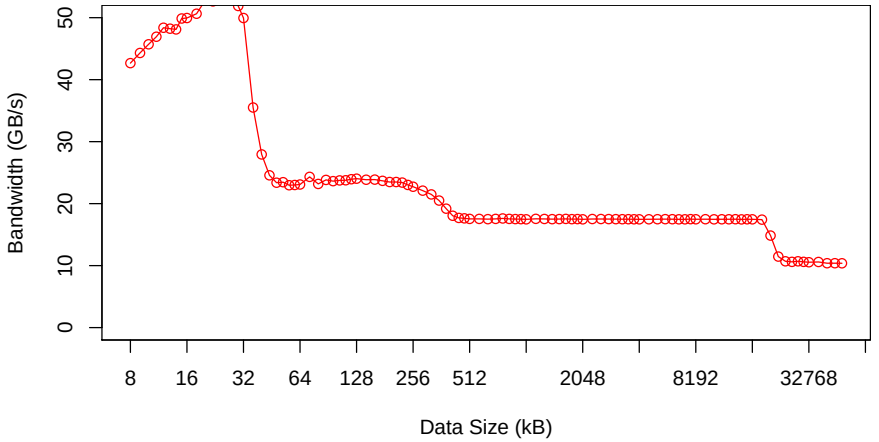
Real hardware architecture



Memory view



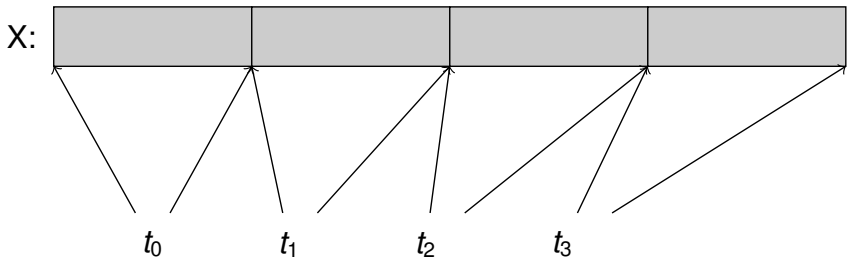
- Simple view of the hardware
- Latency function:
  - Working set size: capacity
  - Access (load *vs.* store)
  - Locality (stride)
  - Data *state*
  - Number of threads accessing memory: contention



Step 2: Software Representation

OpenMP code

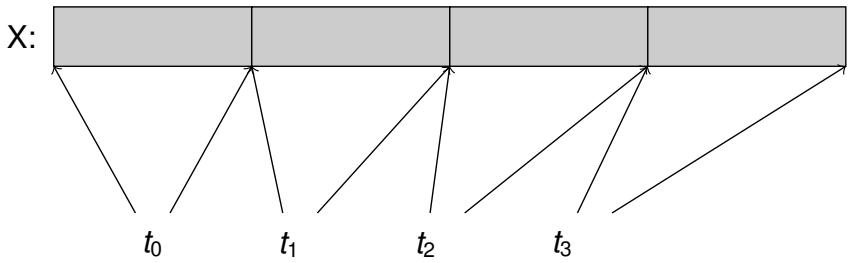
```
void daxpy(double *X, double *Y,
           int size, int a) {
    int i;
#pragma omp parallel for private (i)
    for (i=0; i<size; i++) {
        Y[i] = a*X[i]+Y[i];
    }
}
```



Step 2: Software Representation

OpenMP code

```
void daxpy(double *X, double *Y,
           int size, int a) {
    int i;
#pragma omp parallel for private (i)
    for (i=0; i<size; i++) {
        Y[i] = a*X[i]+Y[i];
    }
}
```



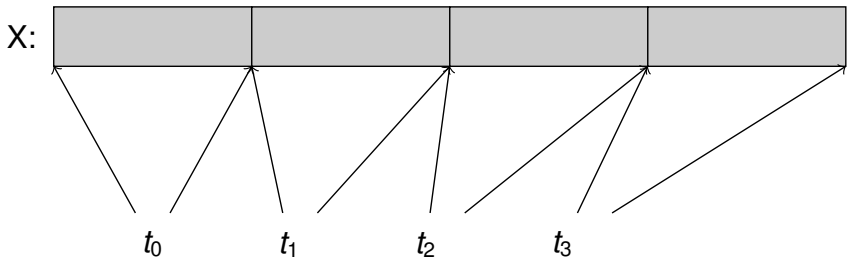
OpenMP chunks access

```
double X[N] ,Y[N];
#pragma omp parallel
int t = omp_get_thread_num();
int T = omp_get_num_threads();
read  ([t*N/T  : (t + 1)*N/T- 1 : 8], X);
read  ([t*N/T  : (t + 1)*N/T- 1 : 8], Y);
write ([t*N/T  : (t + 1)*N/T- 1 : 8], Y);
```

Step 2: Software Representation

OpenMP code

```
void daxpy(double *X, double *Y,
           int size, int a) {
    int i;
#pragma omp parallel for private (i)
    for (i=0; i<size; i++) {
        Y[i] = a*X[i]+Y[i];
    }
}
```



OpenMP chunks access

```
double X[N] ,Y[N];
#pragma omp parallel
int t = omp_get_thread_num();
int T = omp_get_num_threads();
read  ([t*N/T  : (t + 1)*N/T- 1 : 8], X);
read  ([t*N/T  : (t + 1)*N/T- 1 : 8], Y);
write ([t*N/T  : (t + 1)*N/T- 1 : 8], Y);
```

Corresponding representation

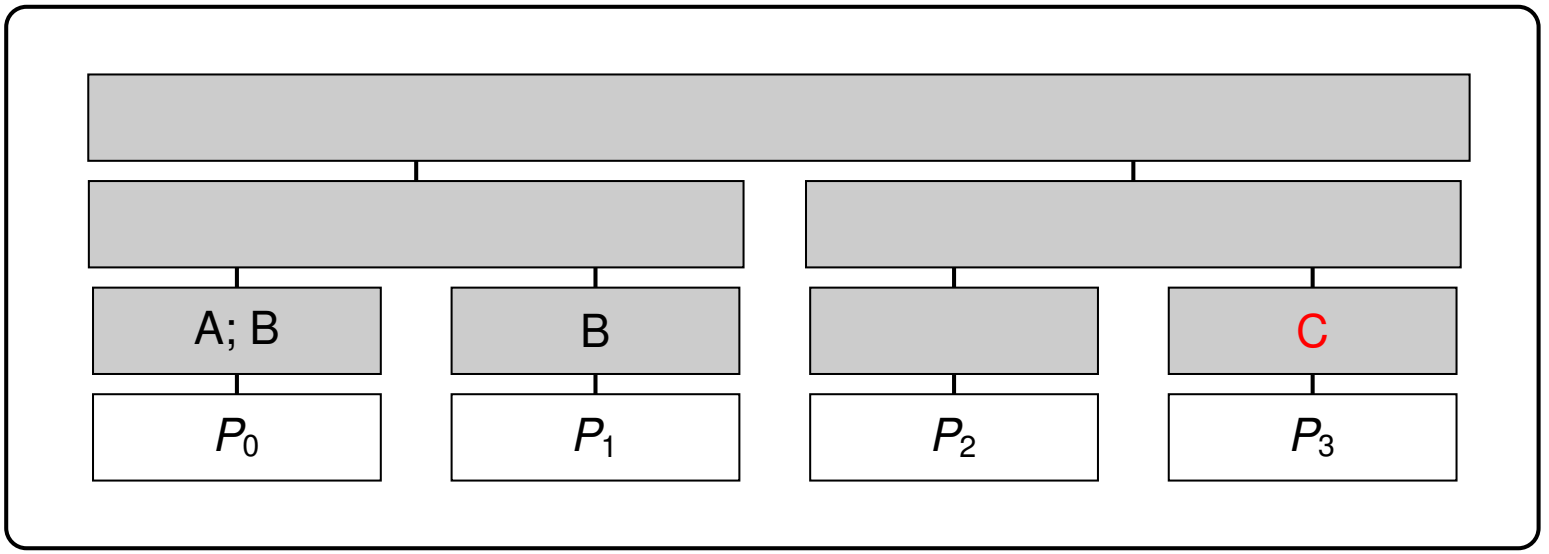
```
daxpy:
    read(f, X)
    read(f, Y)        write(f, Y)
```

Step 3: Memory State

Chunk state is global:

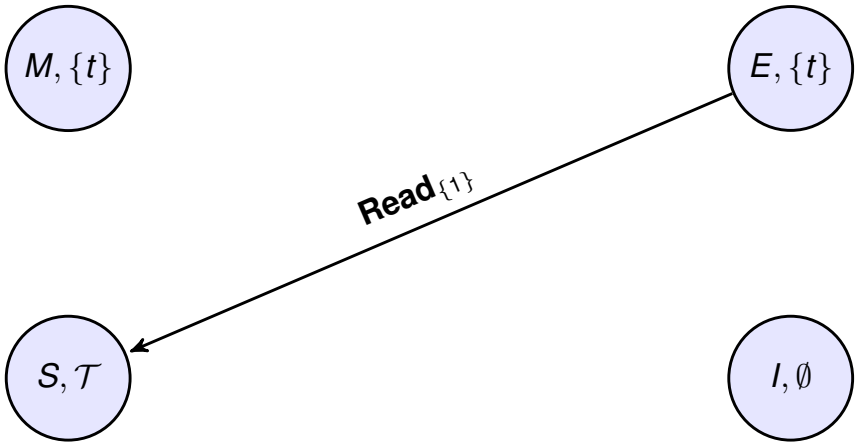
- $M_{\{t\}}$ chunk modified in the cache of thread t
- $E_{\{t\}}$ clean chunk exclusively in the cache of thread t
- $S_{\mathcal{T}}$ clean chunk in caches of all threads $\in \mathcal{T}$
- I chunk not present in any cache

Example:



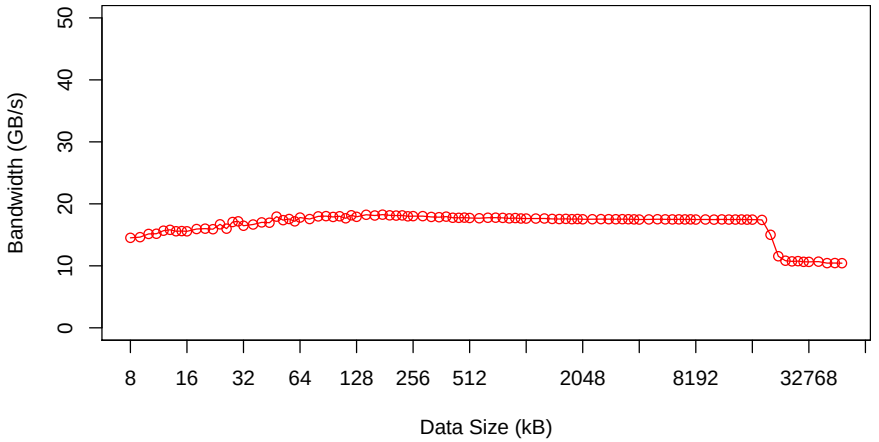
A.state = $E_{\{0\}}$ B.state = $S_{\{0,1\}}$ C.state = $M_{\{3\}}$ D.state = I

Step 3: Memory Automaton (2)

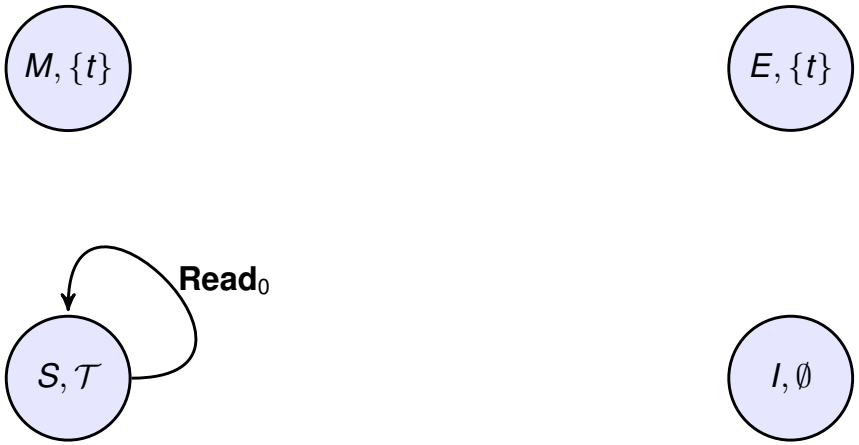


1. A.state = $E_{\{0\}} \Rightarrow \mathbf{Read}_{\{1\}} \Rightarrow$ A.state = $S_{\{0,1\}}$
2. A.state = $S_{\{0,1\}} \Rightarrow \mathbf{Read}_{\{1\}} \Rightarrow$ A.state = $S_{\{0,1\}}$

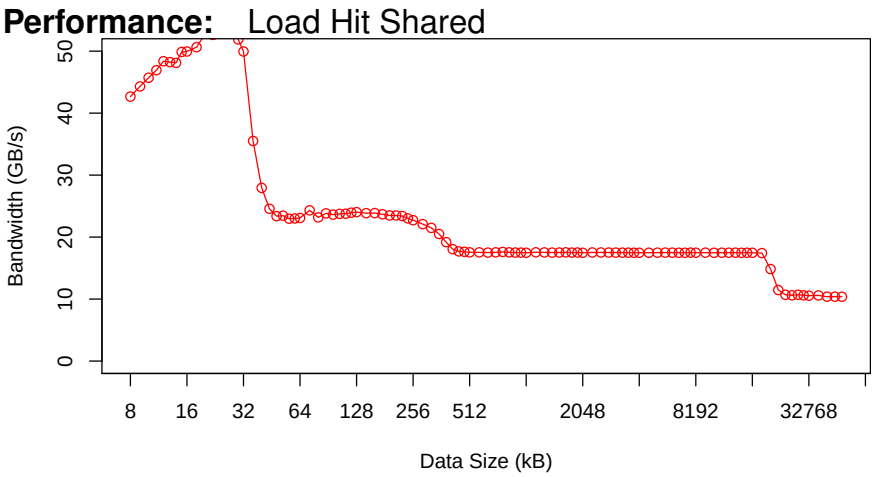
Performance: Load Miss Exclusive



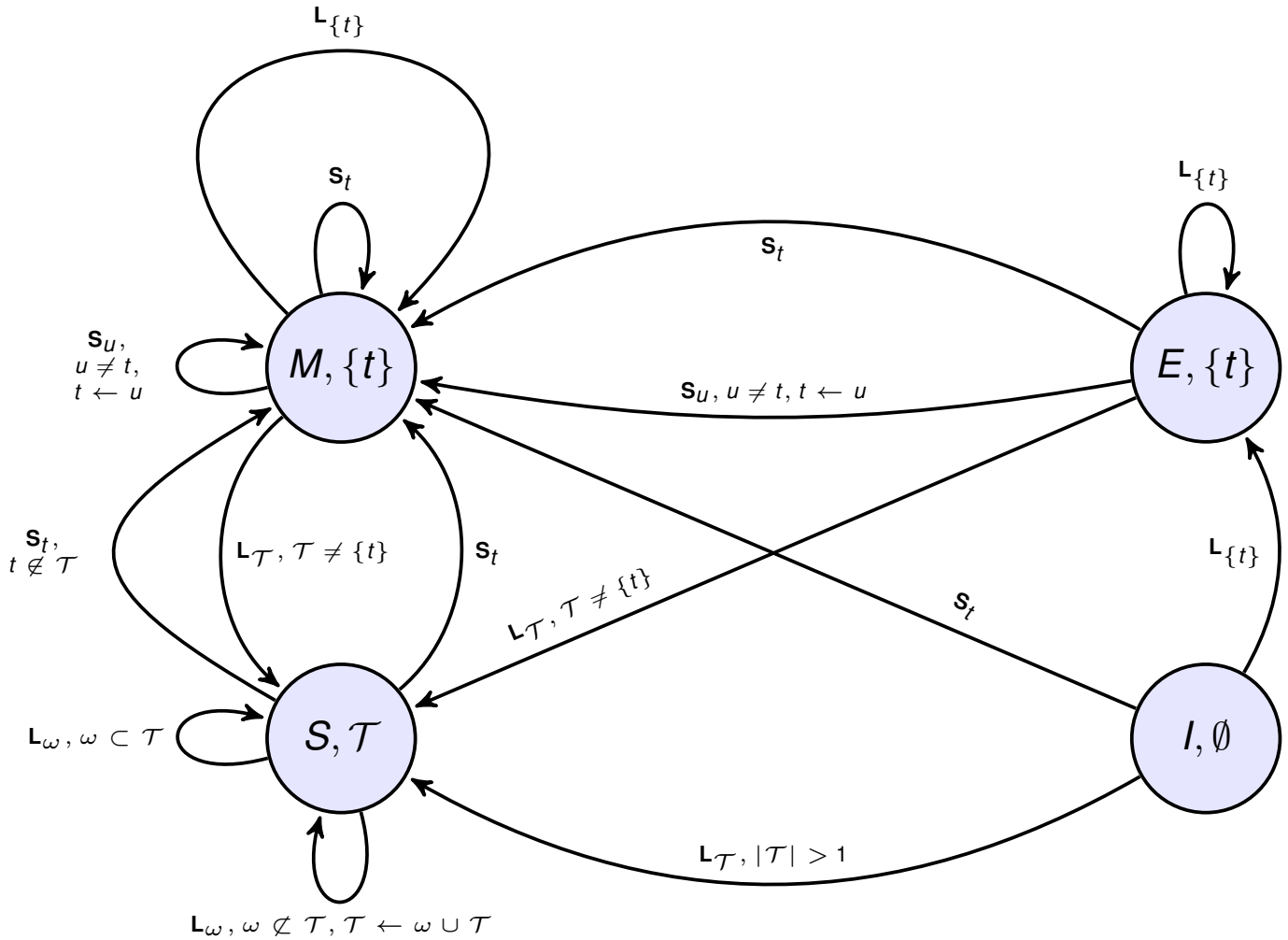
Step 3: Memory Automaton (2)



1. A.state = $E_{\{0\}} \Rightarrow \mathbf{Read}_{\{1\}} \Rightarrow$ A.state = $S_{\{0,1\}}$
2. A.state = $S_{\{0,1\}} \Rightarrow \mathbf{Read}_{\{0\}} \Rightarrow$ A.state = $S_{\{0,1\}}$



Step 3: Memory Automaton (2)

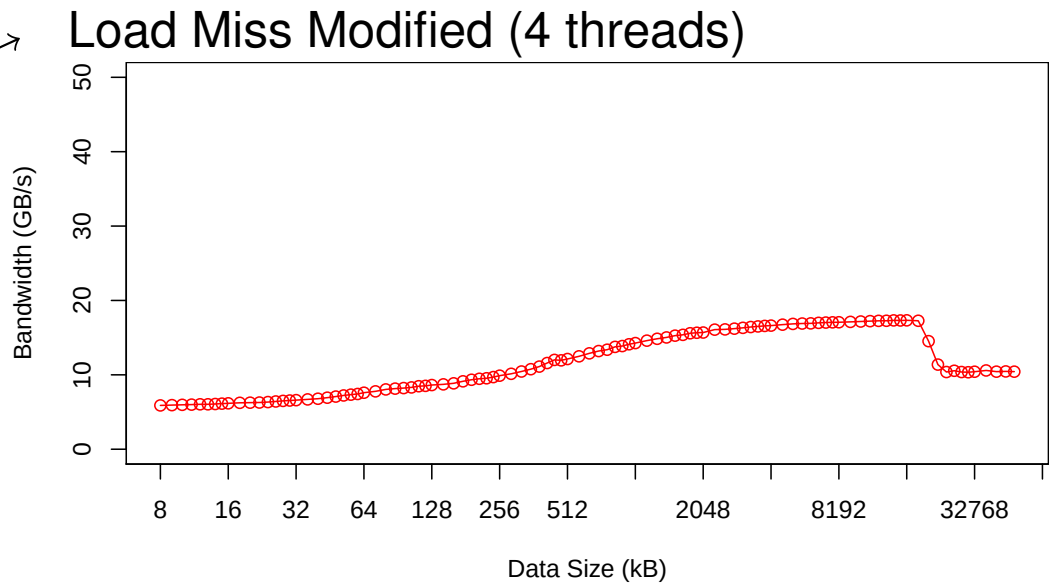


Step 4: Time prediction

Example: DAXPY

With 4 computing threads
Vector size: 8 MB
Initial states: $\forall c \in \mathcal{C}$, c.state $= M_{\{0\}}$

1. read(f, X)
2. read(f, Y)
3. write(f, Y)

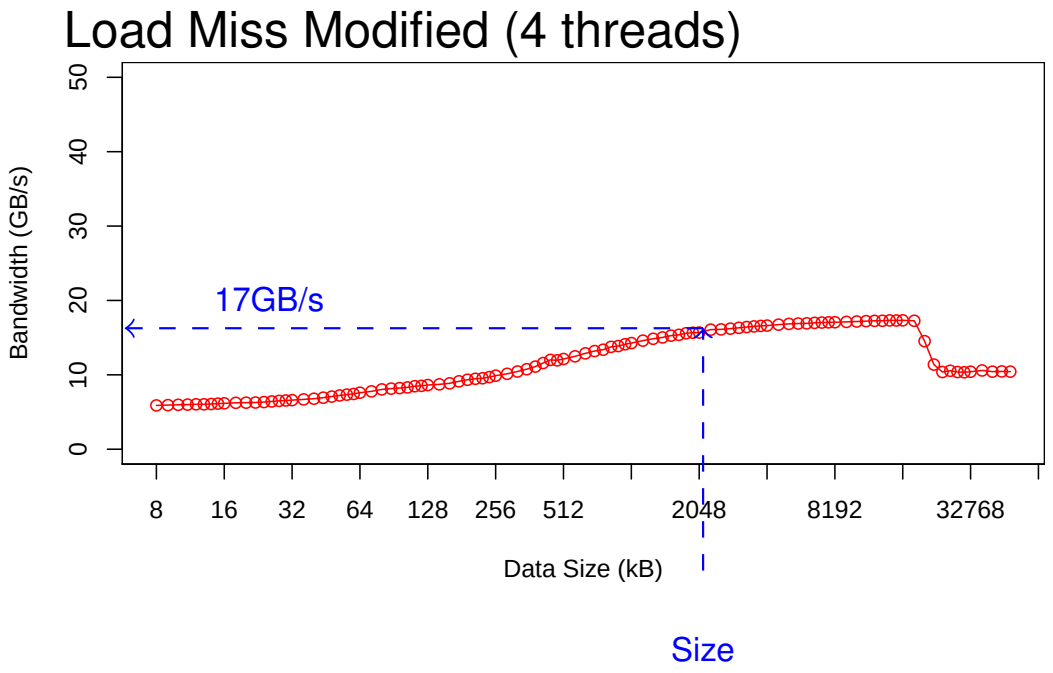


Step 4: Time prediction

Example: DAXPY

With 4 computing threads
Vector size: 8 MB
Initial states: $\forall c \in \mathcal{C}$, c.state $= M_{\{0\}}$

1. read(f, X) $\quad \frac{2\ MB}{17\ GB/s} = 0.11 msec$
2. read(f, Y)
3. write(f, Y)



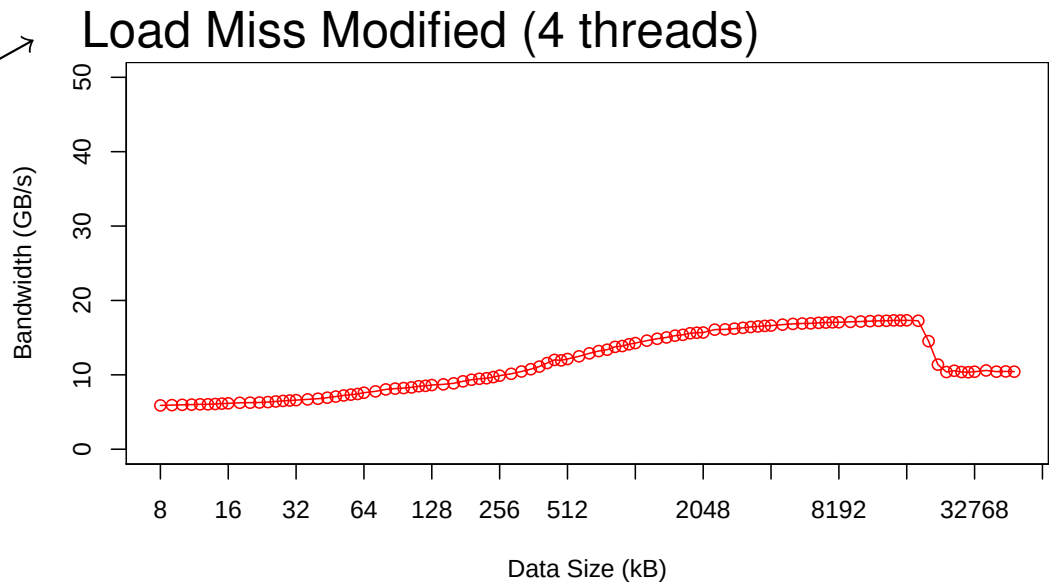
Step 4: Time prediction

Example: DAXPY

With 4 computing threads
Vector size: 8 MB
Initial states: $\forall c \in \mathcal{C}$, c.state $= M_{\{0\}}$

1. read(f, X)
2. read(f, Y)
3. write(f, Y)

$$\frac{2\ MB}{17\ GB/s} = 0.11 msec$$

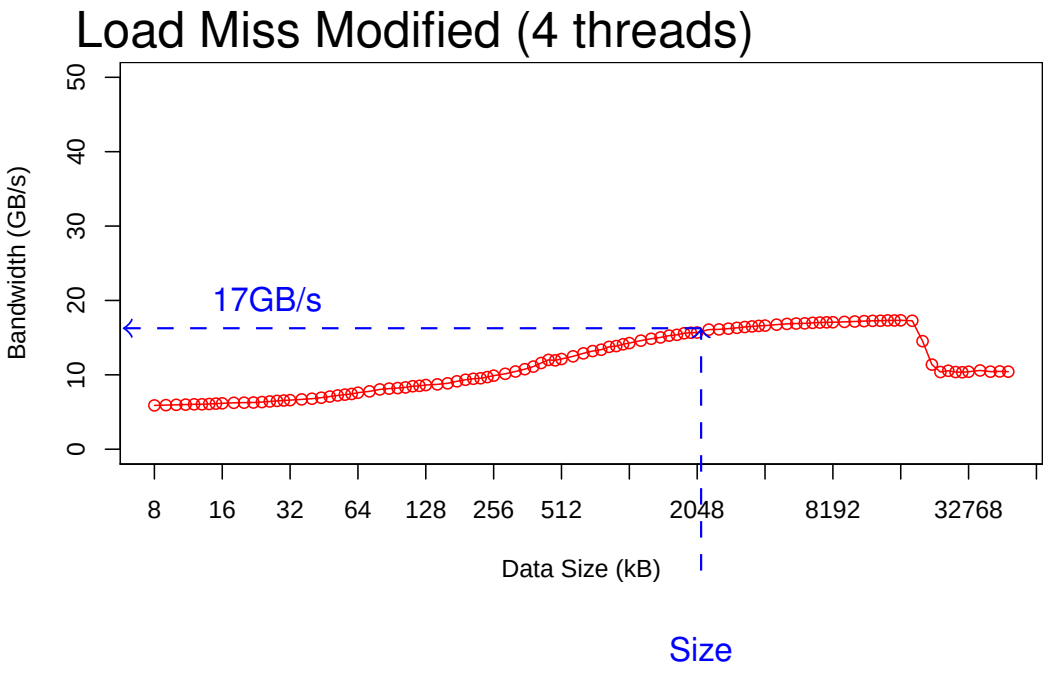


Step 4: Time prediction

Example: DAXPY

With 4 computing threads
Vector size: 8 MB
Initial states: $\forall c \in \mathcal{C}$, c.state $= M_{\{0\}}$

1. read(f, X) $\quad \frac{2\ MB}{17\ GB/s} = 0.11 msec$
2. read(f, Y) $\quad \frac{2\ MB}{17\ GB/s} = 0.11 msec$
3. write(f, Y)

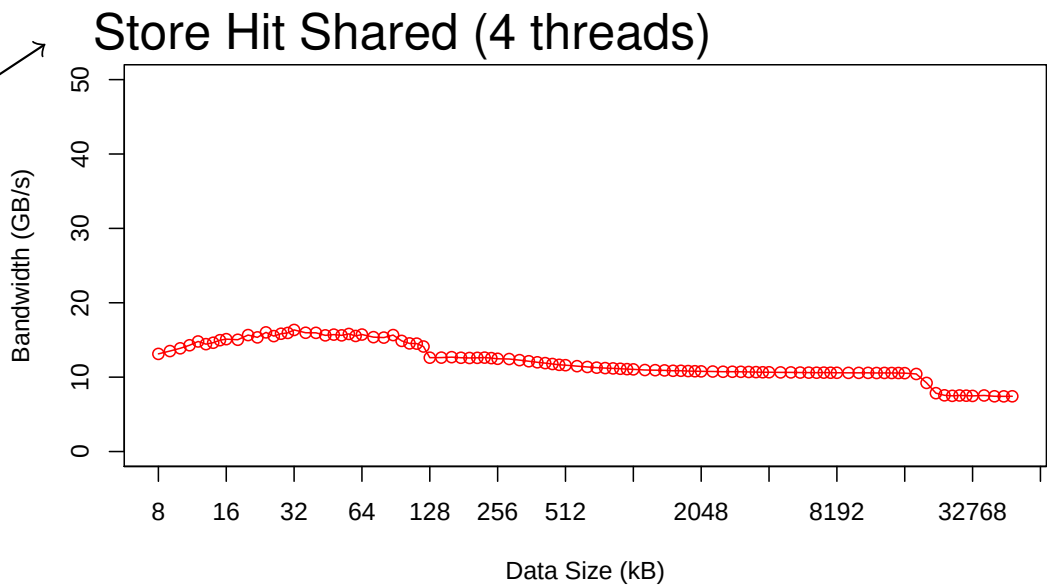


Step 4: Time prediction

Example: DAXPY

With 4 computing threads
Vector size: 8 MB
Initial states: $\forall c \in \mathcal{C}$, c.state $= M_{\{0\}}$

1. read(f, X) $\quad \frac{2\ MB}{17\ GB/s} = 0.11 msec$
2. read(f, Y) $\quad \frac{2\ MB}{17\ GB/s} = 0.11 msec$
3. write(f, Y)

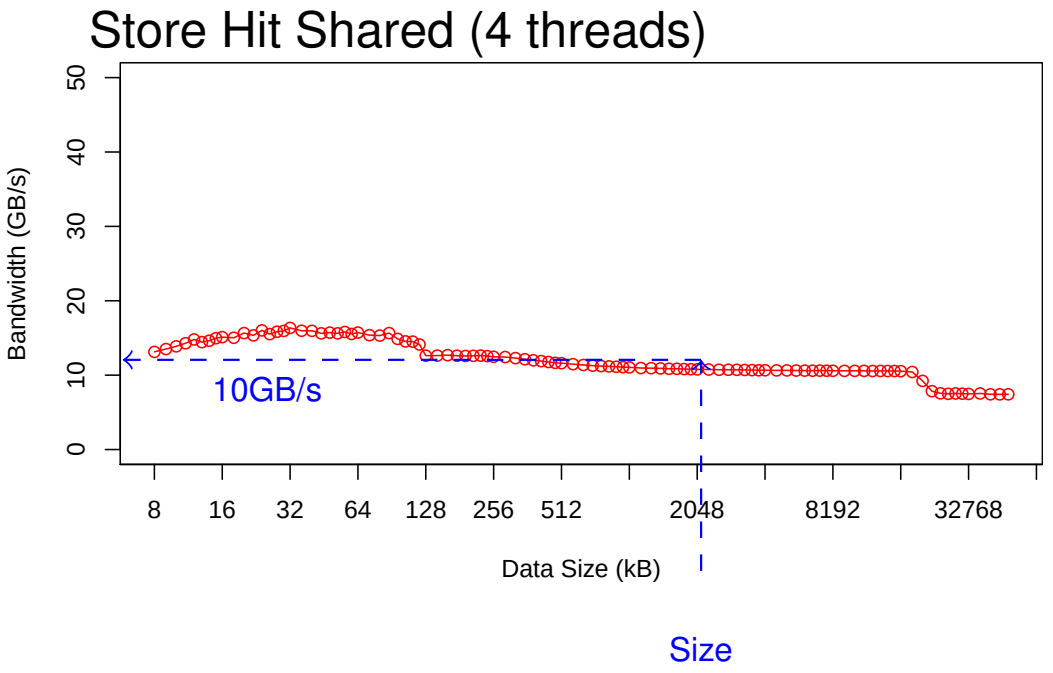


Step 4: Time prediction

Example: DAXPY

With 4 computing threads
Vector size: 8 MB
Initial states: $\forall c \in \mathcal{C}$, c.state $= M_{\{0\}}$

1. read(f, X) $\quad \frac{2\ MB}{17\ GB/s} = 0.11 msec$
2. read(f, Y) $\quad \frac{2\ MB}{17\ GB/s} = 0.11 msec$
3. write(f, Y) $\quad \frac{2\ MB}{10\ GB/s} = 0.2 msec$



Step 4: Time prediction

Example: DAXPY

With 4 computing threads
Vector size: 8 MB
Initial states: $\forall c \in \mathcal{C}$, c.state $= M_{\{0\}}$

1. read(f, X) $\frac{2\ MB}{17\ GB/s} = 0.11 msec$
2. read(f, Y) $\frac{2\ MB}{17\ GB/s} = 0.11 msec$
3. write(f, Y) $\frac{2\ MB}{10\ GB/s} = 0.2 msec$

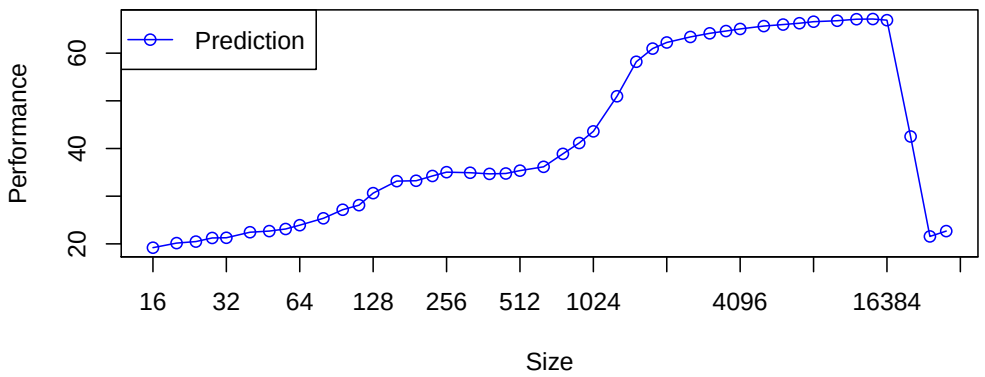
$$\text{time} = \text{MAX}\Big(0.11 + 0.11, 0.2\Big)$$

Step 4: Time prediction

Example: DAXPY

With 4 computing threads
Vector size: 8 MB
Initial states: $\forall c \in \mathcal{C}$, c.state $= M_{\{0\}}$

1. read(f, X) $\frac{2\ MB}{17\ GB/s} = 0.11 msec$
2. read(f, Y) $\frac{2\ MB}{17\ GB/s} = 0.11 msec$
3. write(f, Y) $\frac{2\ MB}{10\ GB/s} = 0.2 msec$



Actual Performance Prediction

Strong scalability prediction:

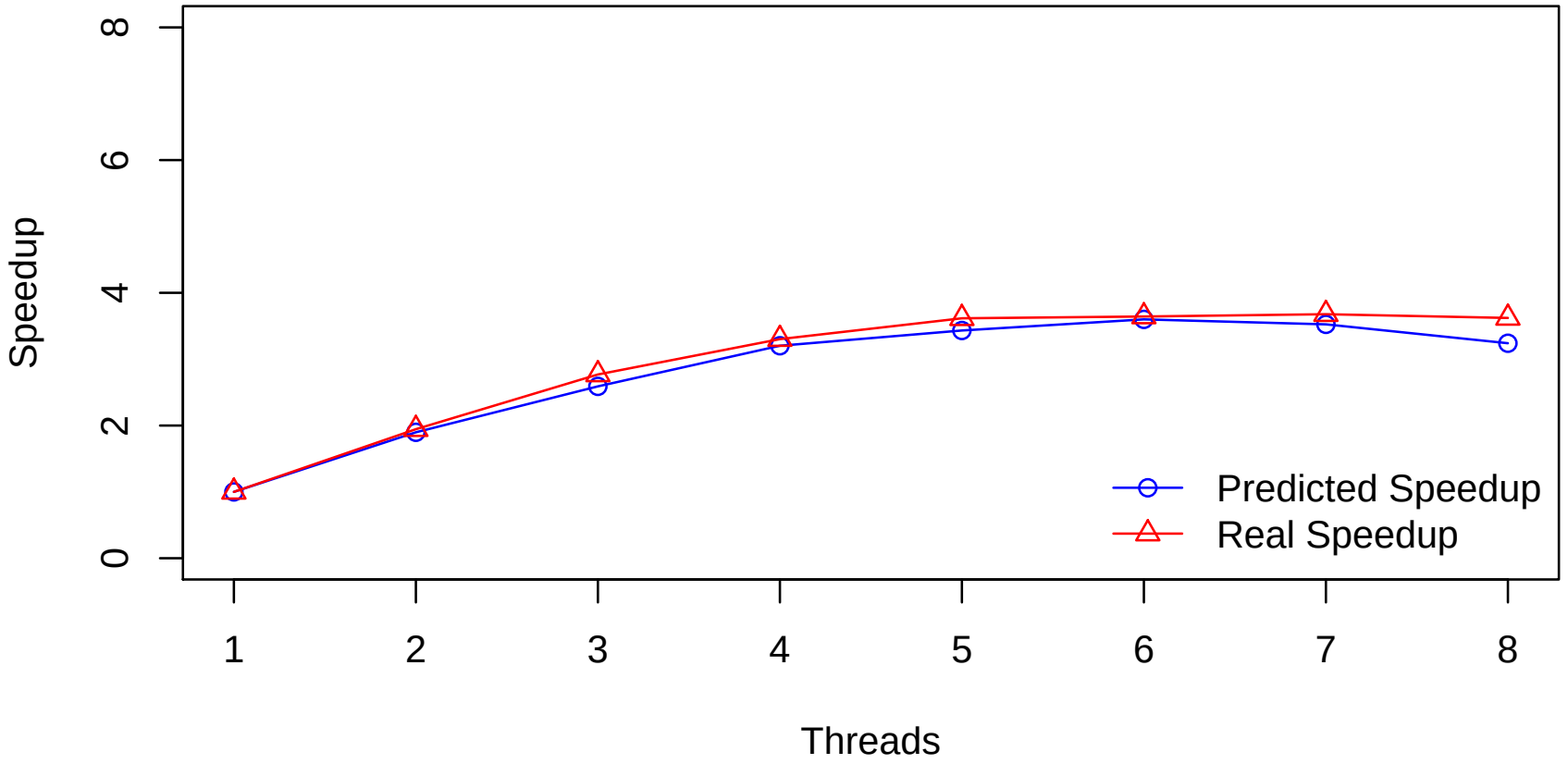


Figure: Dotproduct 64 MB

Actual Performance Prediction

Strong scalability prediction:

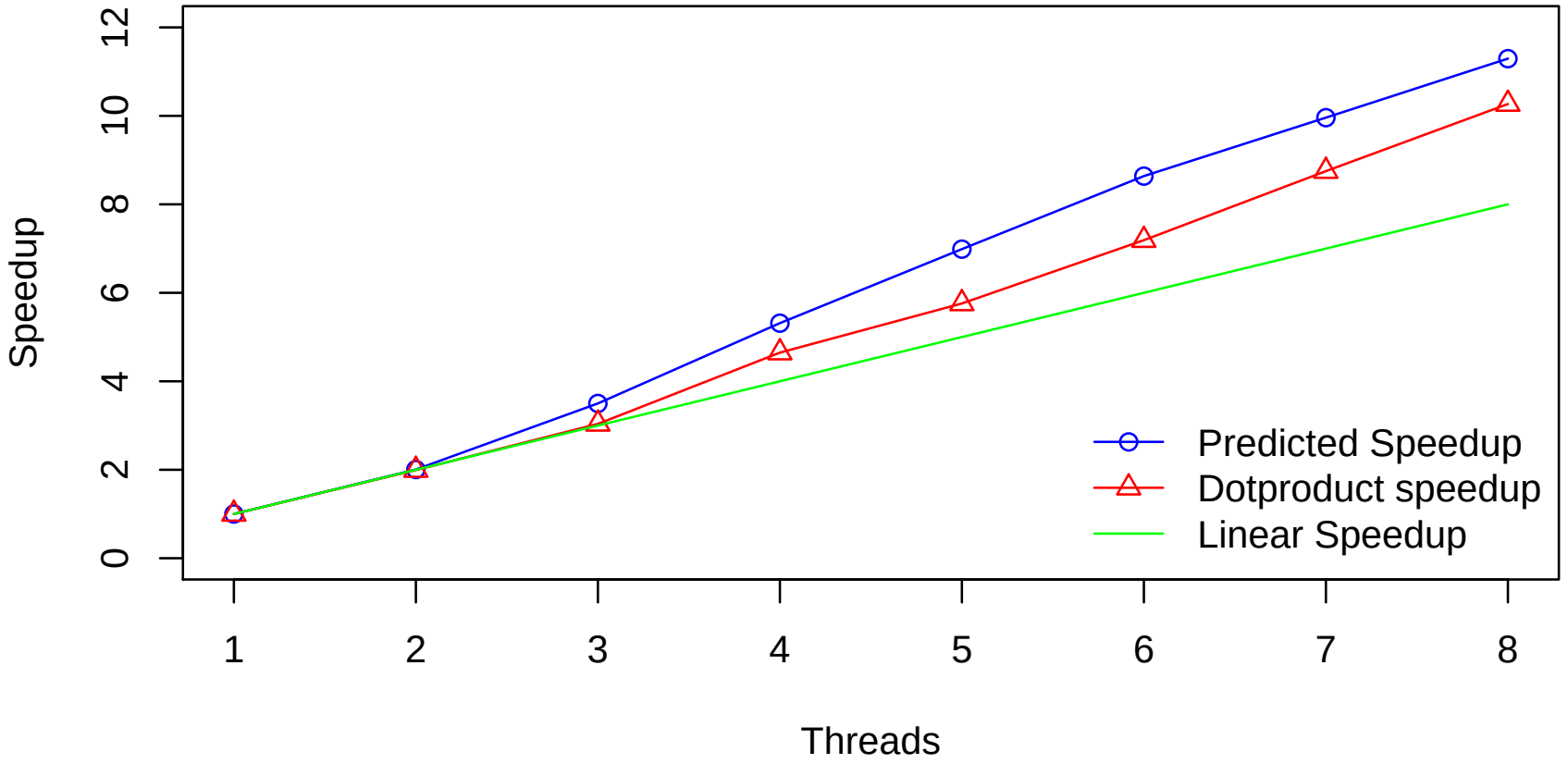


Figure: Dotproduct 1 MB

Actual Performance Prediction

Strong scalability prediction:

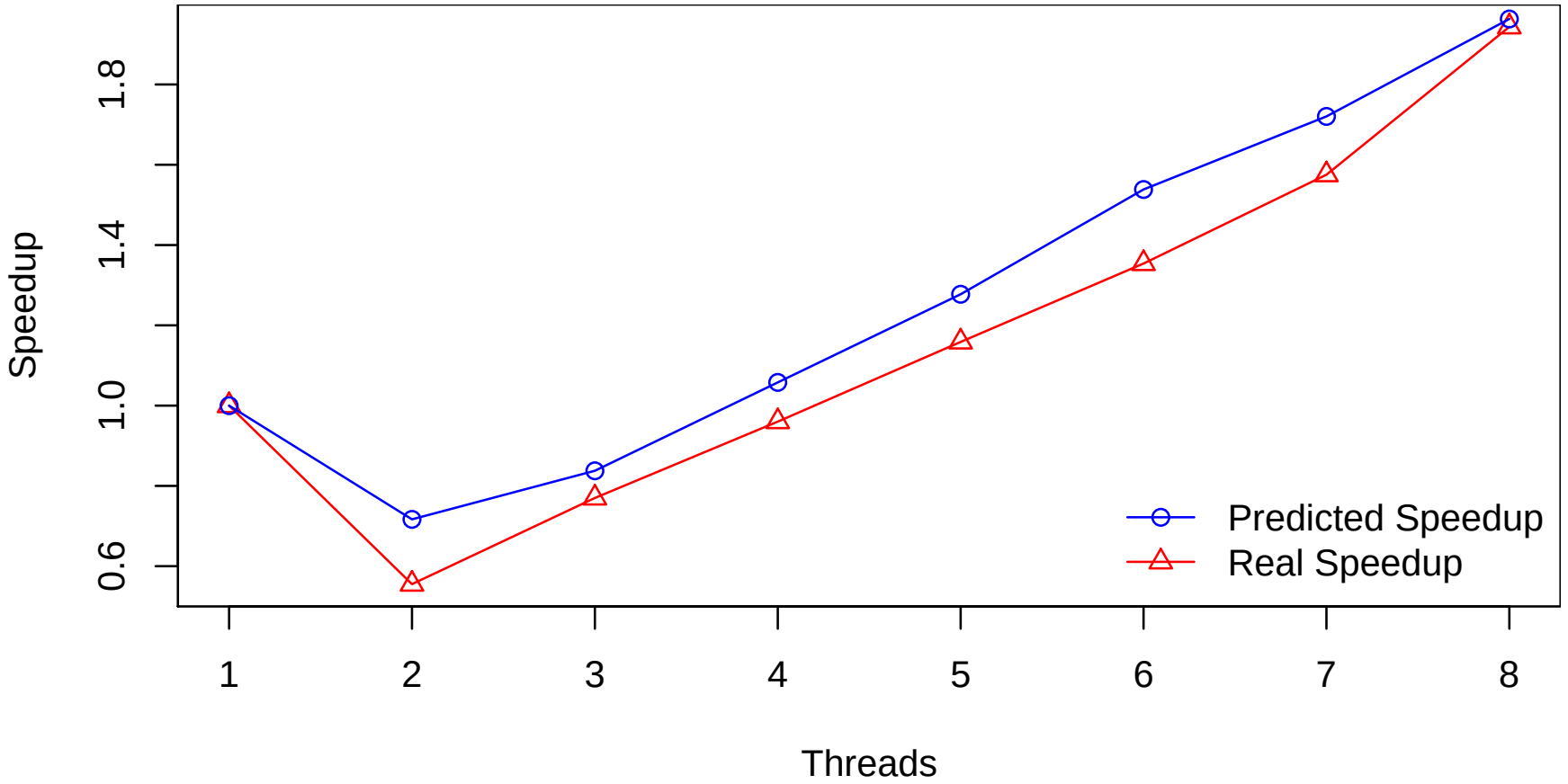


Figure: Dotproduct 64 kB

Actual Performance Prediction

Strong scalability prediction:

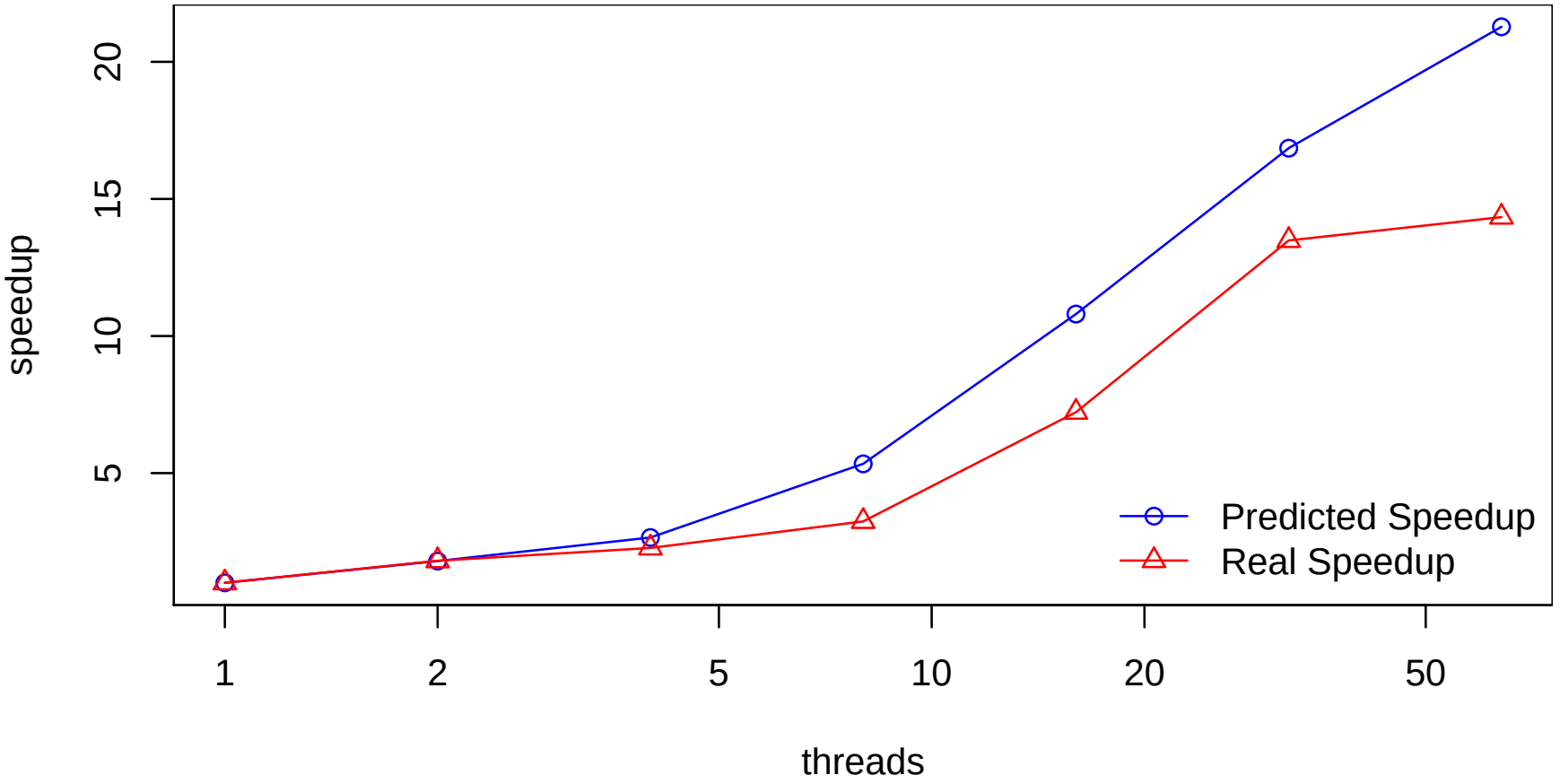


Figure: Dotproduct 1 MB on Xeon Phi

Actual Performance Prediction: Krylov algorithms

- Iterative solvers
- Sparse matrix with dense blocks
- More-complex applications
- Basic blocks are BLAS 1 and 2
- Krylov methods execution time:
  - Number of iterations
  - Time of every iteration

CG:

$r = b$;
$p = r$;
$n_r = (r|r)$;
$x = 0$;
while $n_r > \epsilon$ **do**
  $A_p = A \times p$; // BLAS 2
  $\alpha = n_r/(p|A_p)$; // BLAS 1
  $x = x + \alpha \times p$;
  $r = r - \alpha \times A_p$;
  $n_{r1} = (r|r)$;
  $\beta = n_{r1}/n_r$;
  $p = r + \beta \times p$;
  $n_r = n_{r1}$
end

Actual Performance Prediction: Krylov algorithms

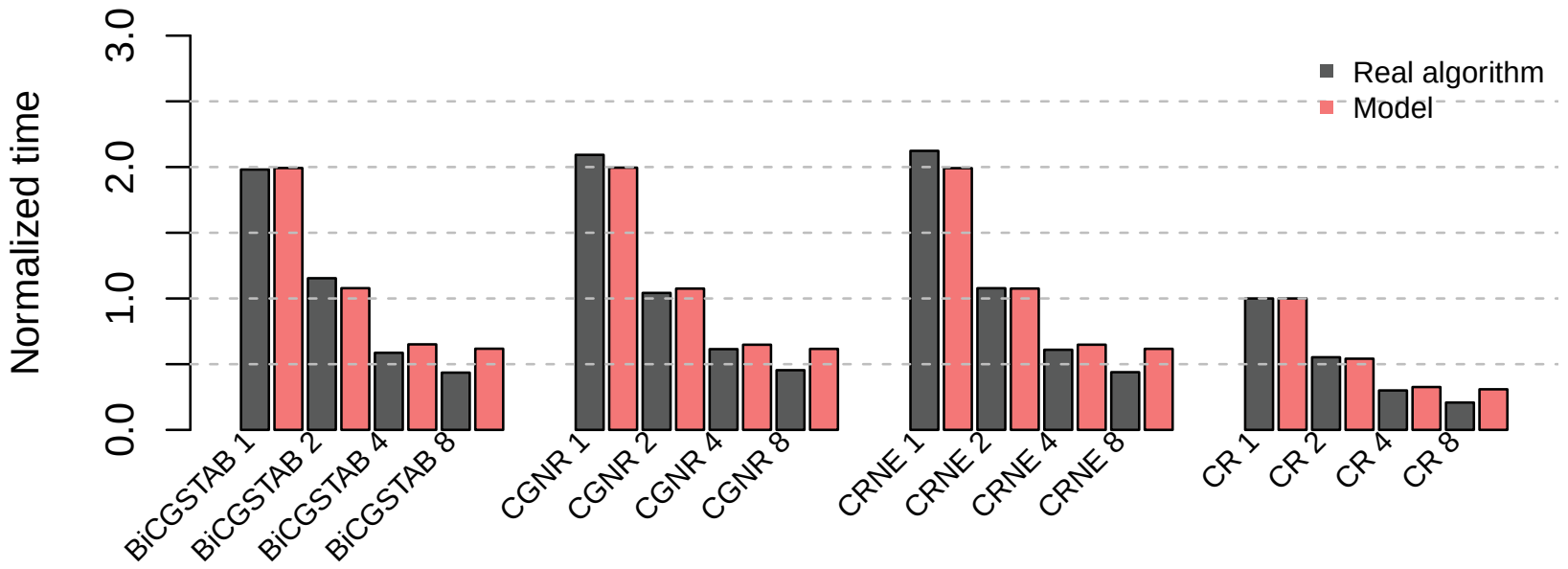


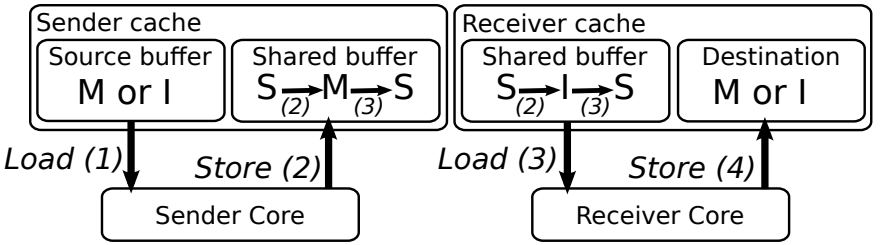
Figure: Reference time: CR, sequential time of the fastest algorithm

Sandy-Bridge processor

Applying the model to MPI shared memory communications

MPI communication anatomy:

Pipelined copies



- Bottleneck identified
- Optimization opportunities with non-temporal or explicite flush instructions
- Workshop publication[2]

[2] B. Putigny, B. Ruelle, B. Goglin: Analysis of MPI Shared-Memory Communication Performance from a Cache Coherence Perspective. In Workshop on Parallel and Distributed Scientific and Engineering Computing. 2014

Memory Model: Summary

	stack model	proposed model	simulators
Compulsory	✓	✓	✓
Capacity	✓	✓	✓
Contention	✗	✓	✓
Coherence	✗	✓	✓
Conflict	✗	✗	✓

Achievements:

- Successfully applied to several Krylov algorithms
- Also applied to model MPI communications through shared memory

Strengths:

- Predicts time
- Automatic calibration to new architectures

[2] Submitted to: The 2014 International Conference on High Performance Computing & Simulation

Outline

Summary

Performance modeling

- Optimization decision
- Better hardware utilization
- Time prediction

Contributions

- Computational modeling
  - Methodologies toward automatic instruction performance retrieval
  - Successfully modeled the SCC
  - Helped modeling the Xeon Phi at Intel's Exascale Computing Research Lab
- Memory Modeling
  - Demonstrated time prediction on QCD mini applications
  - Fast model adaptation thanks to benchmark-based calibration
  - MPI shared memory communications modeled

Short term Perspectives

Memory model

- Automatic extraction of memory access (Step 2)[1]
- Better capacity model: stack
- Use memory traces (*e.g.,* from MAQAO)
- Plug it into a simulator
  - Simulators: memory access
  - Memory model: access cost

[1] Cetus compiler: `http://cetus.ecn.purdue.edu/`

Long term Perspectives

Toward a unified model

- Full hardware model (intra-node)
- Large-scale platforms

Memory model

- Automatic detection of cache coherence issues
- Hardware simulation
  - Given expected performance of a given architecture, predict complex application scalability,
  - Bottleneck detection (contention, coherence ...)

Thank you!