

Complementing User-Level Coarse-Grain Parallelism with Implicit Speculative Parallelism

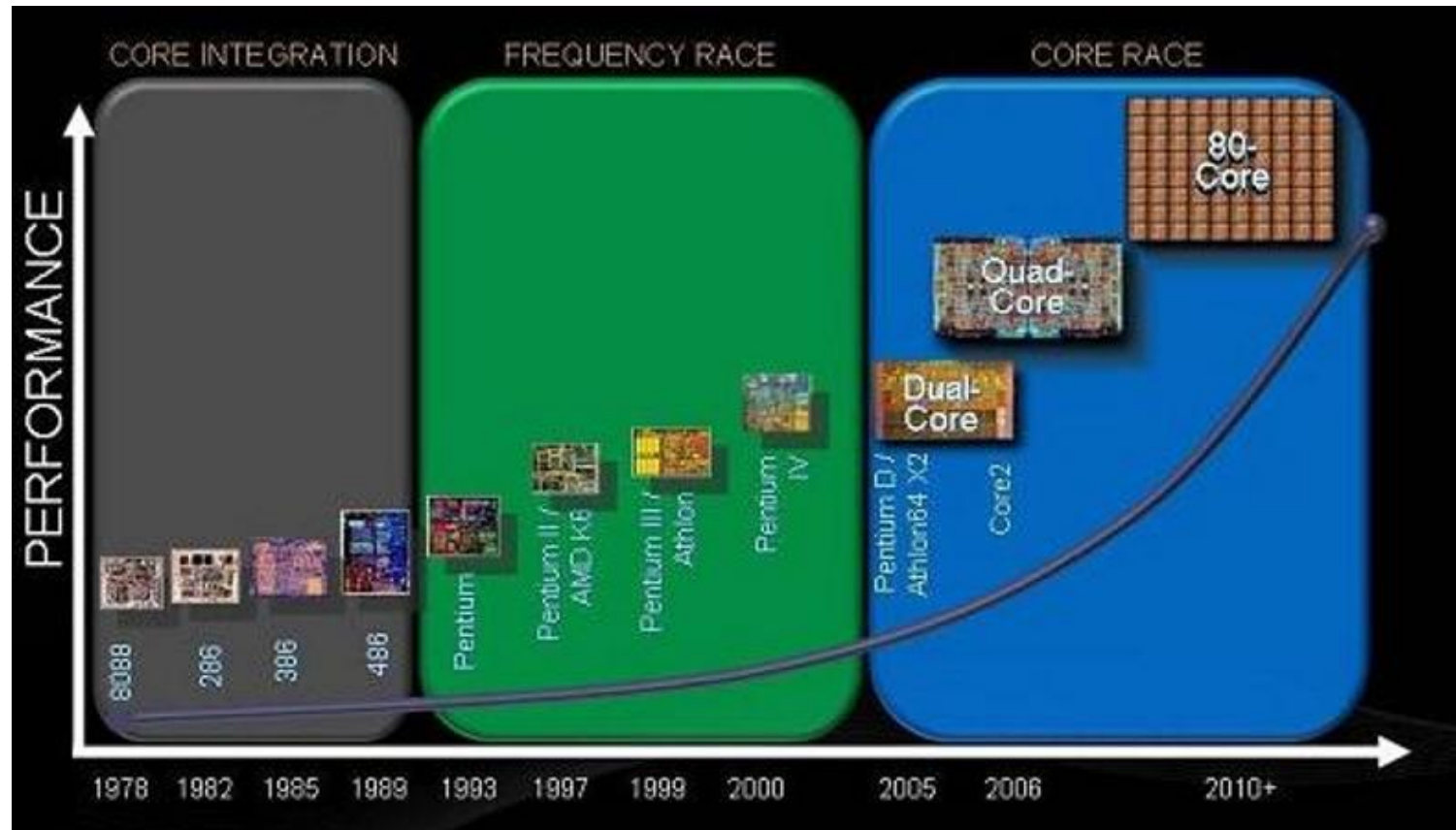
Nikolas Ioannou, Marcelo Cintra

School of Informatics
University of Edinburgh



Introduction

- Multi-cores and many-cores here to stay



Source: Intel

Introduction

- Multi-cores and many-cores are here to stay
- Parallel programming is essential to realize potential
- Focus on coarse-grain parallelism
- Weak or no scaling of some parallel applications
- Can we exploit under-utilized cores to complement coarse-grain parallelism?
 - Nested parallelism in multi-threaded applications
 - Exploit it using implicit speculative parallelism

Contributions

- Evaluation of implicit speculative parallelism on top of explicit parallelism to improve scalability:
 - Improve scalability by 40% on avg.
 - Same energy consumption
- Detailed analysis of multithreaded scalability:
 - Performance bottlenecks
 - Behavior on different input datasets
- Auto-tuning to dynamically select the number of explicit and implicit threads

Outline

- Introduction
- **Motivation**
- Proposal
- Evaluation Methodology
- Results
- Conclusions

Bottlenecks: Large Critical Sections

T_0 T_1 T_2 T_3

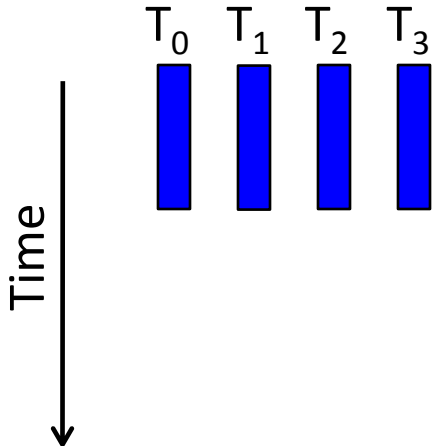
Time
↓

Integer Sort (IS)

NASPB



Bottlenecks: Large Critical Sections

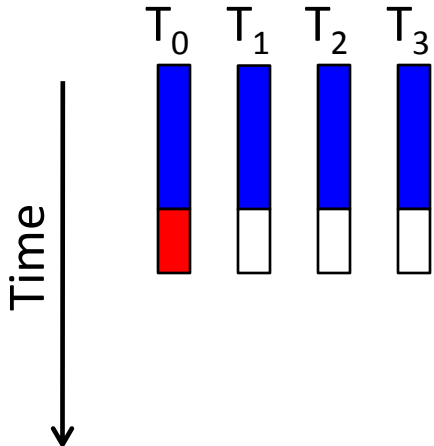


Integer Sort (IS)

NASPB



Bottlenecks: Large Critical Sections

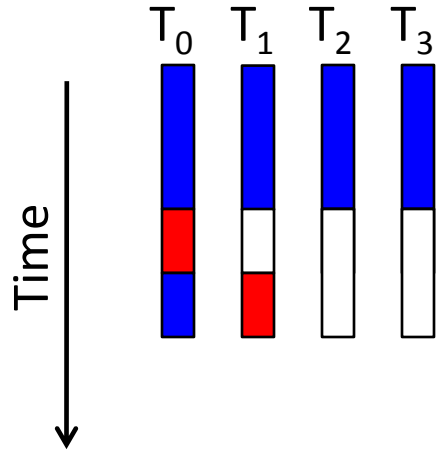


Integer Sort (IS)

NASPB



Bottlenecks: Large Critical Sections

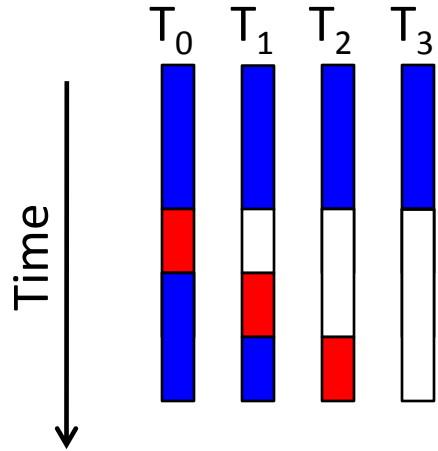


Integer Sort (IS)

NASPB



Bottlenecks: Large Critical Sections

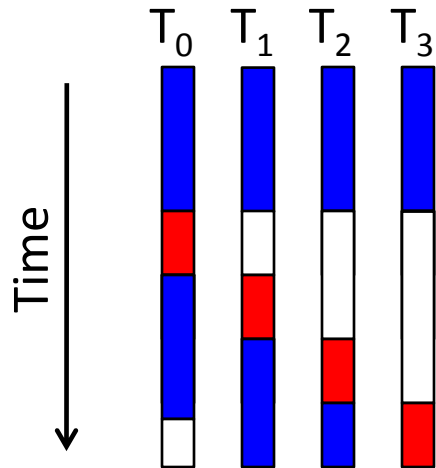


Integer Sort (IS)

NASPB



Bottlenecks: Large Critical Sections

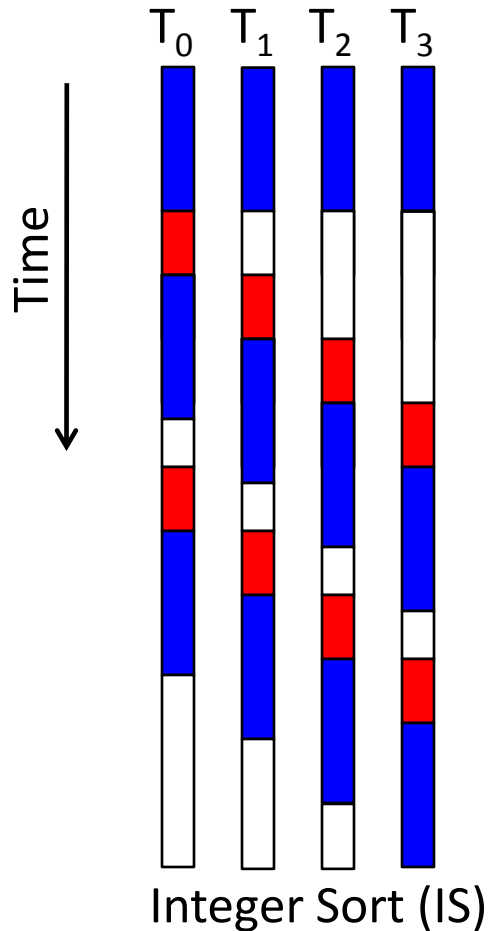


Integer Sort (IS)

NASPB



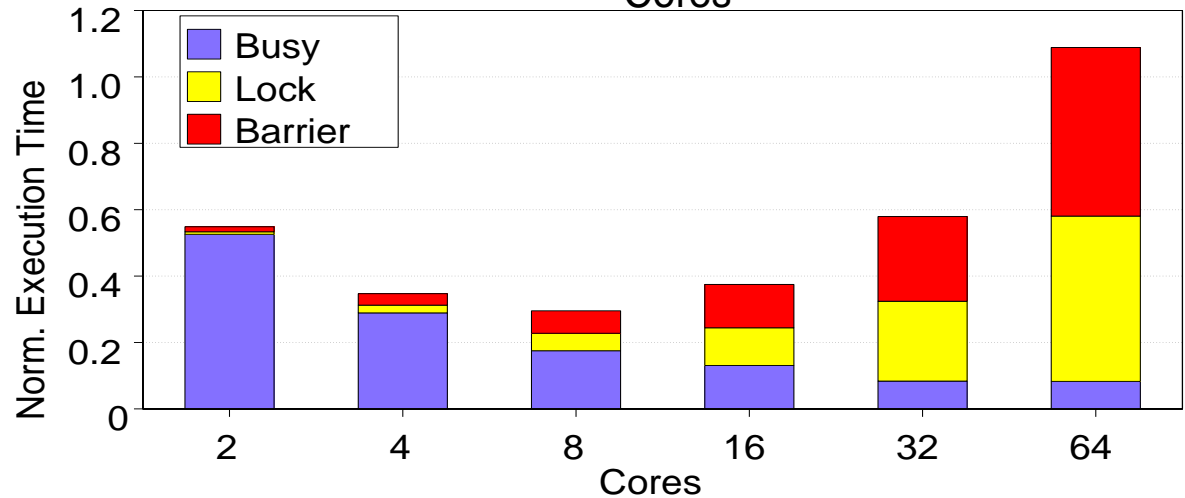
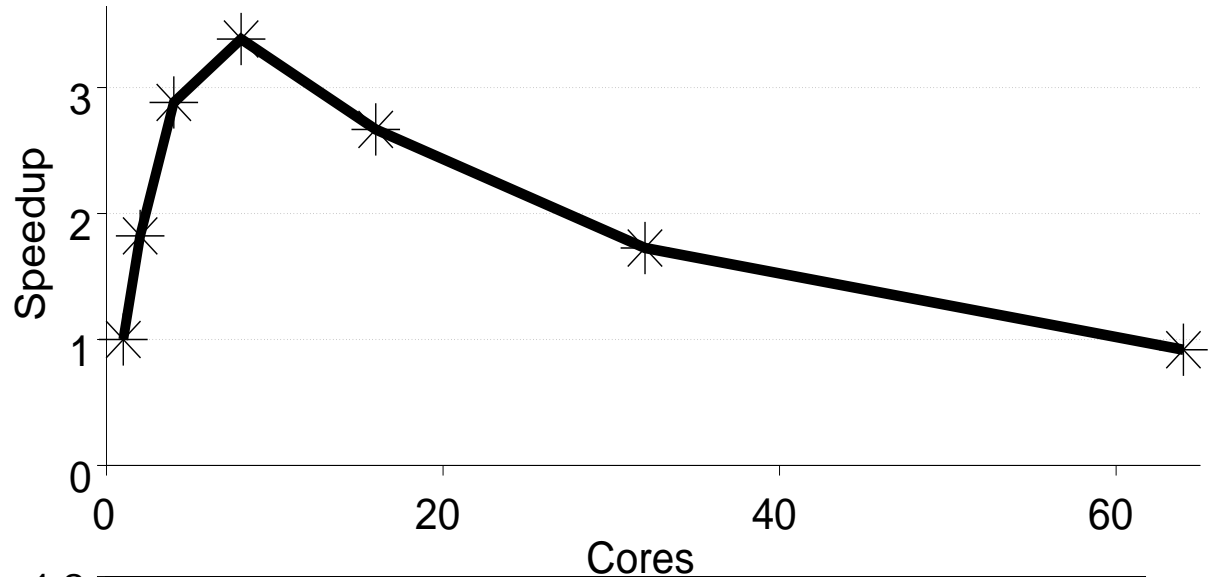
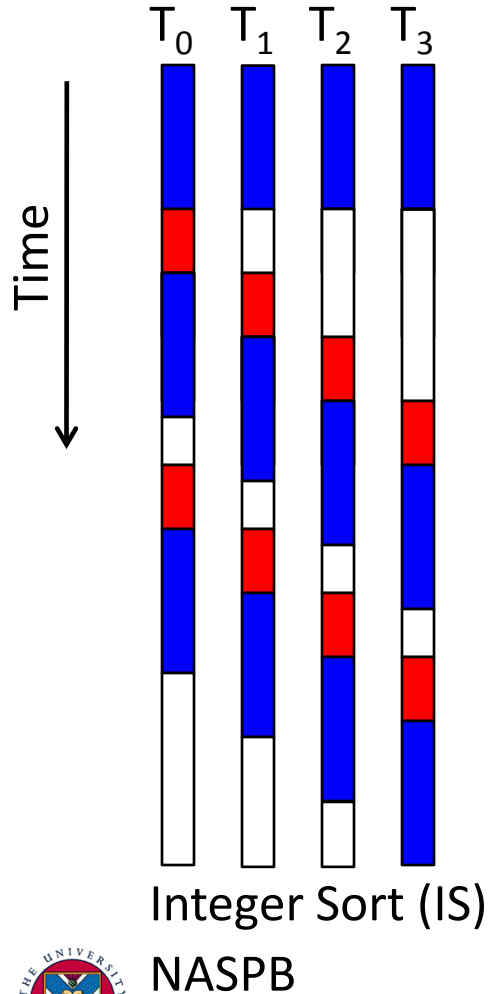
Bottlenecks: Large Critical Sections



NASPB



Bottlenecks: Large Critical Sections



Bottlenecks: Load Imbalance

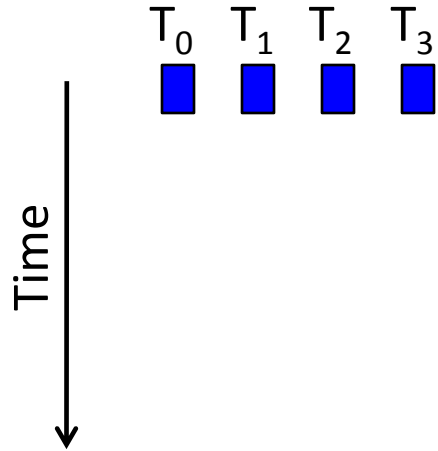
T_0 T_1 T_2 T_3

Time
↓

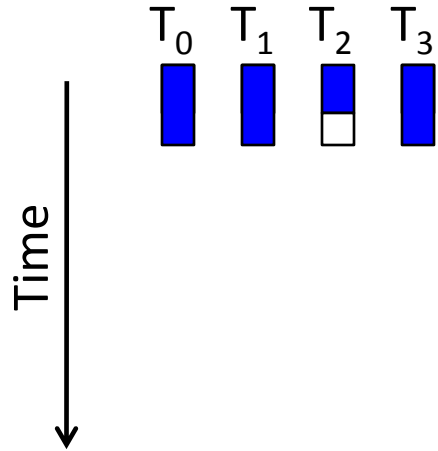
RADIOSITY
SPLASH 2



Bottlenecks: Load Imbalance



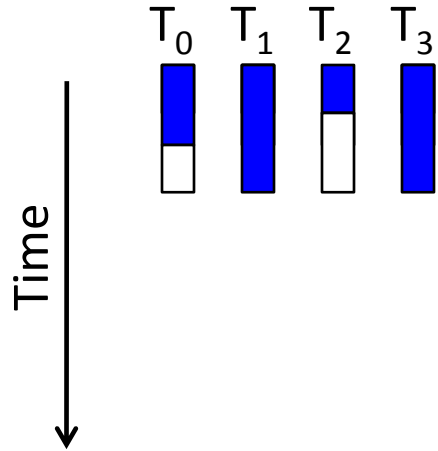
Bottlenecks: Load Imbalance



RADIOSITY
SPLASH 2



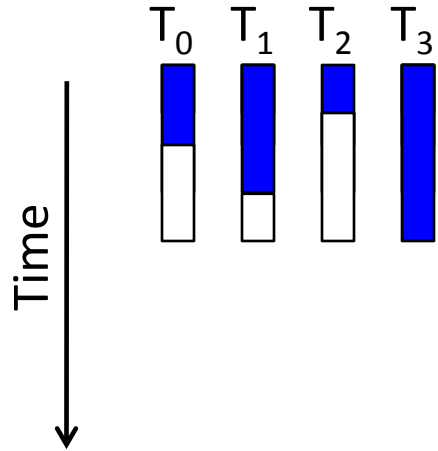
Bottlenecks: Load Imbalance



RADIOSITY
SPLASH 2



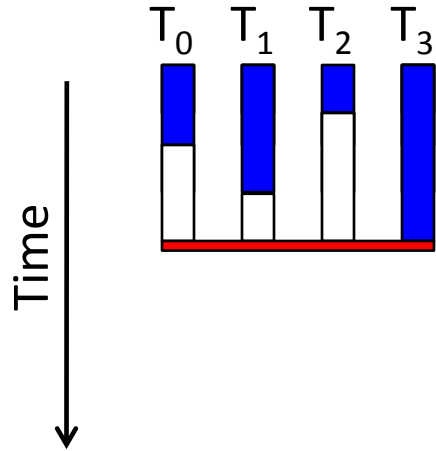
Bottlenecks: Load Imbalance



RADIOSITY
SPLASH 2



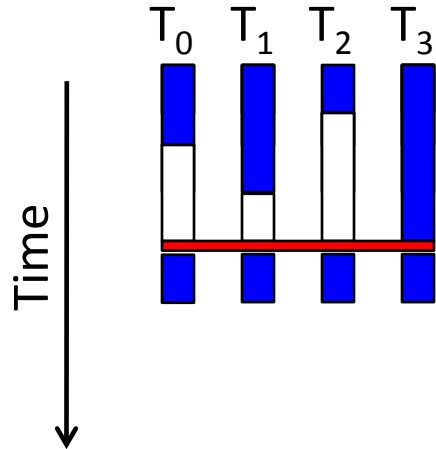
Bottlenecks: Load Imbalance



RADIOSITY
SPLASH 2



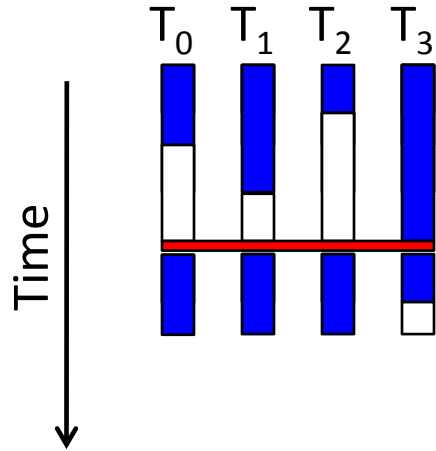
Bottlenecks: Load Imbalance



RADIOSITY
SPLASH 2



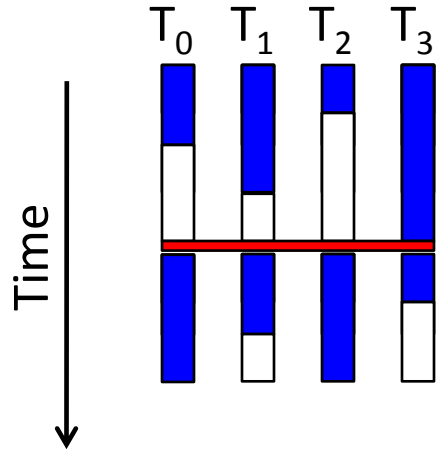
Bottlenecks: Load Imbalance



RADIOSITY
SPLASH 2



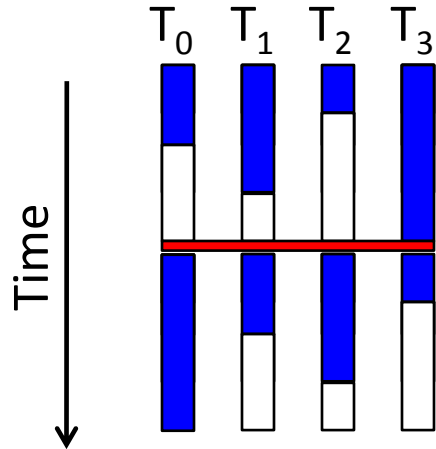
Bottlenecks: Load Imbalance



RADIOSITY
SPLASH 2



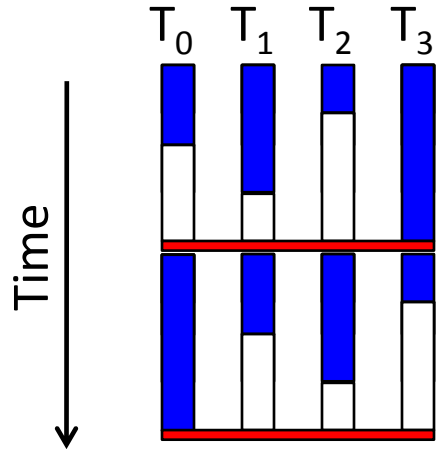
Bottlenecks: Load Imbalance



RADIOSITY
SPLASH 2



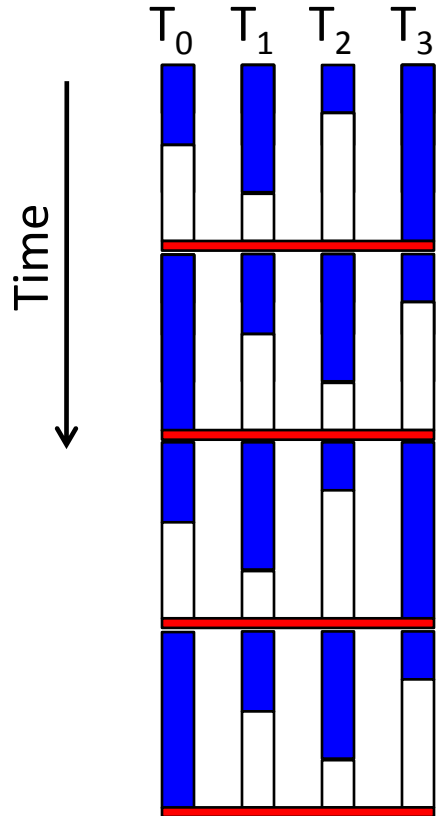
Bottlenecks: Load Imbalance



RADIOSITY
SPLASH 2



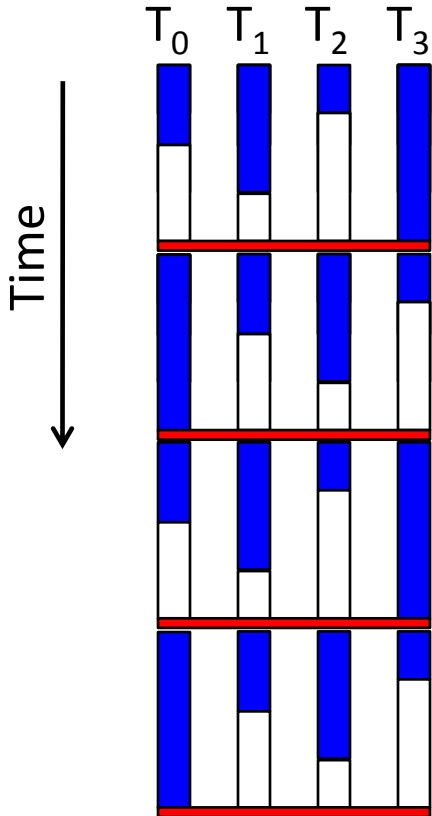
Bottlenecks: Load Imbalance



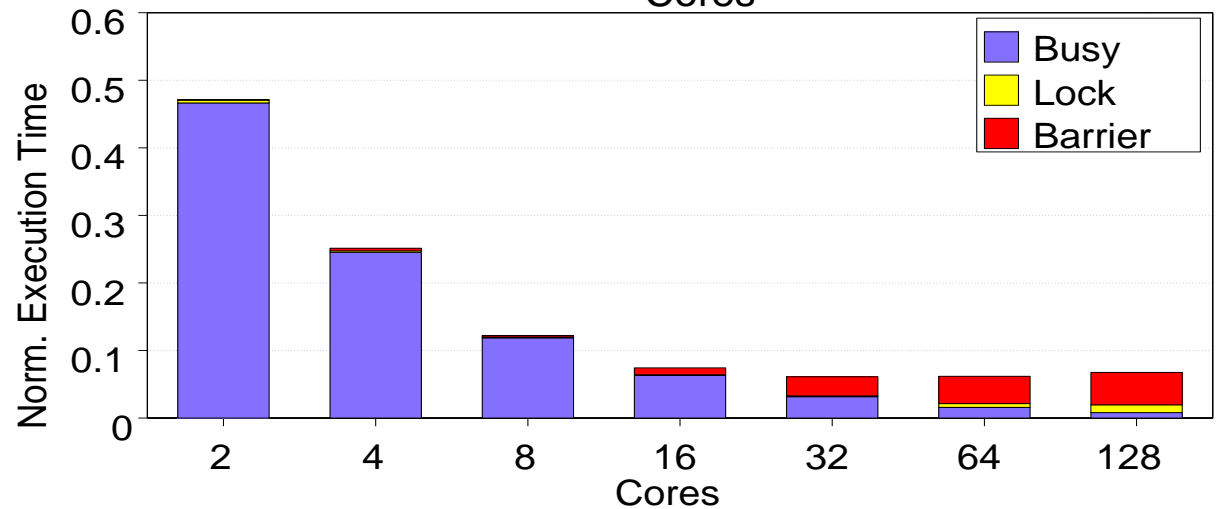
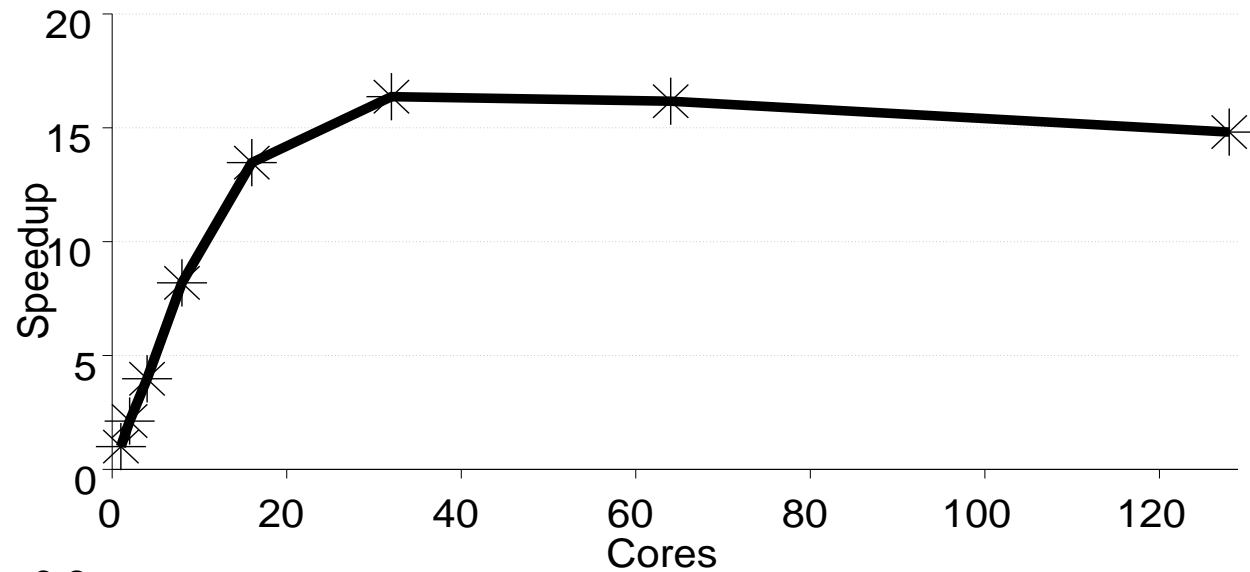
RADIOSITY
SPLASH 2



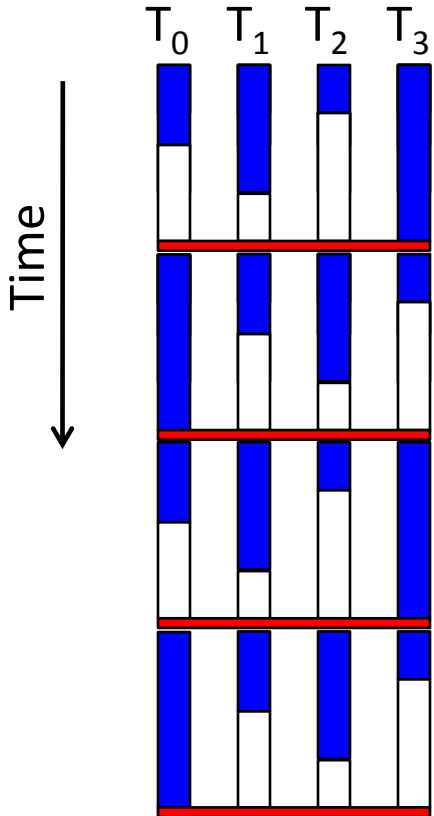
Bottlenecks: Load Imbalance



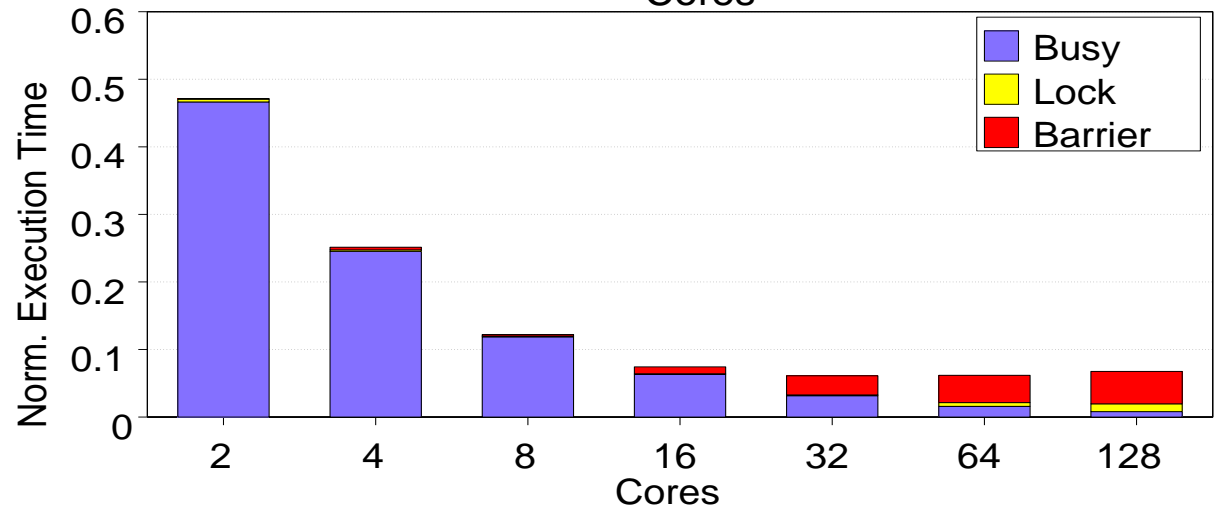
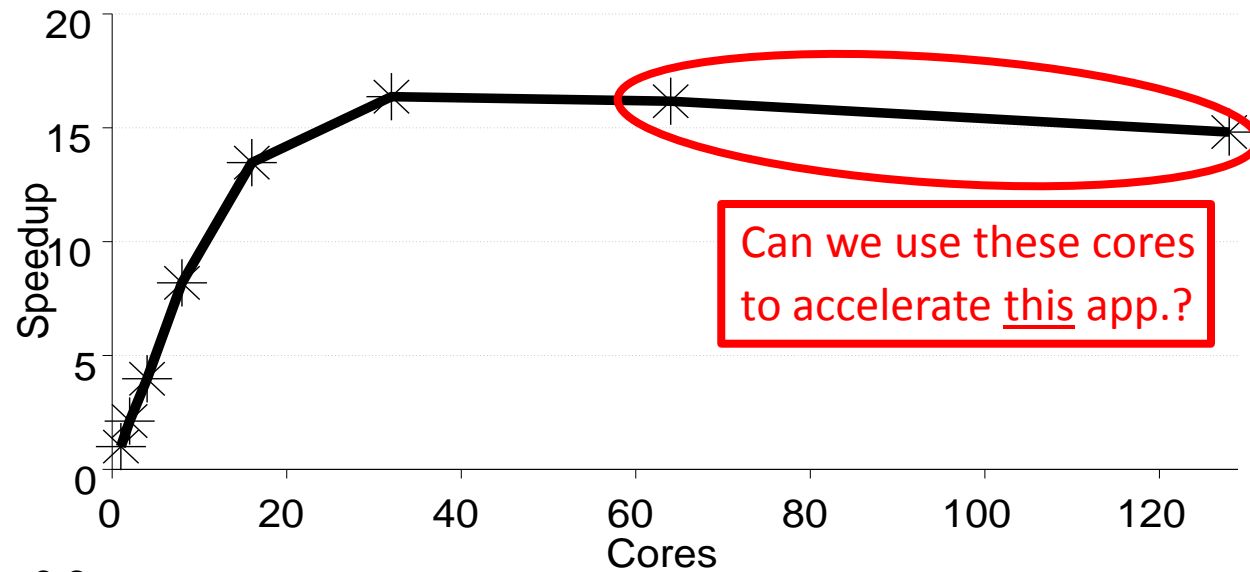
RADIOSSITY
SPLASH 2



Bottlenecks: Load Imbalance



RADIOSSITY
SPLASH 2



Outline

- Introduction
- Motivation
- **Proposal**
- Evaluation Methodology
- Results
- Low power nested parallelism
- Conclusions

Proposal

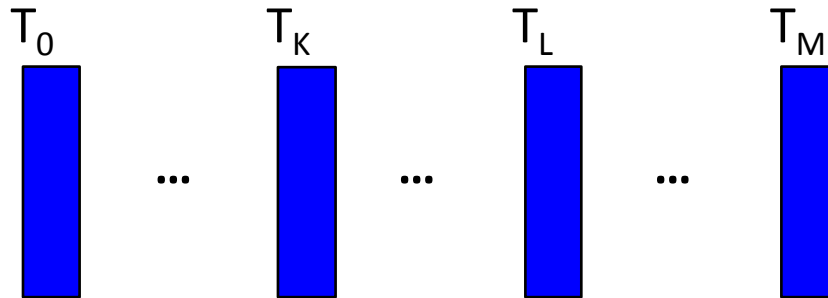
- Programming:
 - Users explicitly parallelize code
 - Tradeoff development time for performance gains
- Architecture and Compiler:
 - Exploit fine-grain parallelism on top of user threads
 - Thread-Level Speculation (TLS) within each user thread
- Hardware:
 - Support both explicit and implicit threads simultaneously in a nested fashion

Proposal

```
#pragma omp parallel for
for(j = 0; j < M; ++j) {
    ...
    for(i = 0; i < N; ++i)
    {
        ... = A[L[i]] + ...
        ...
        A[K[i]] = ...
    }
    ...
}
```

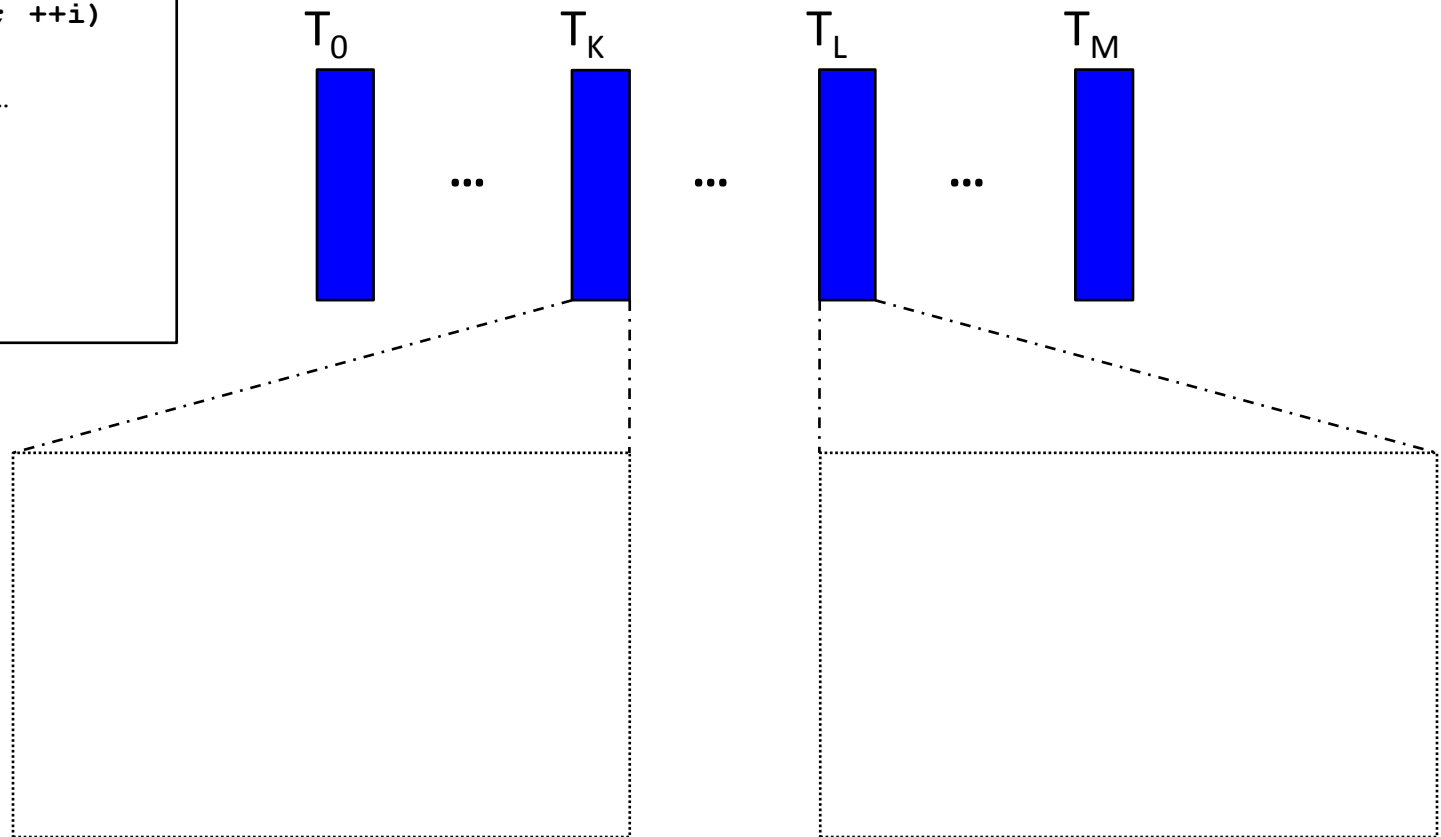
Proposal

```
#pragma omp parallel for
for(j = 0; j < M; ++j) {
  ...
  for(i = 0; i < N; ++i)
  {
    ... = A[L[i]] + ...
    ...
    A[K[i]] = ...
  }
  ...
}
```



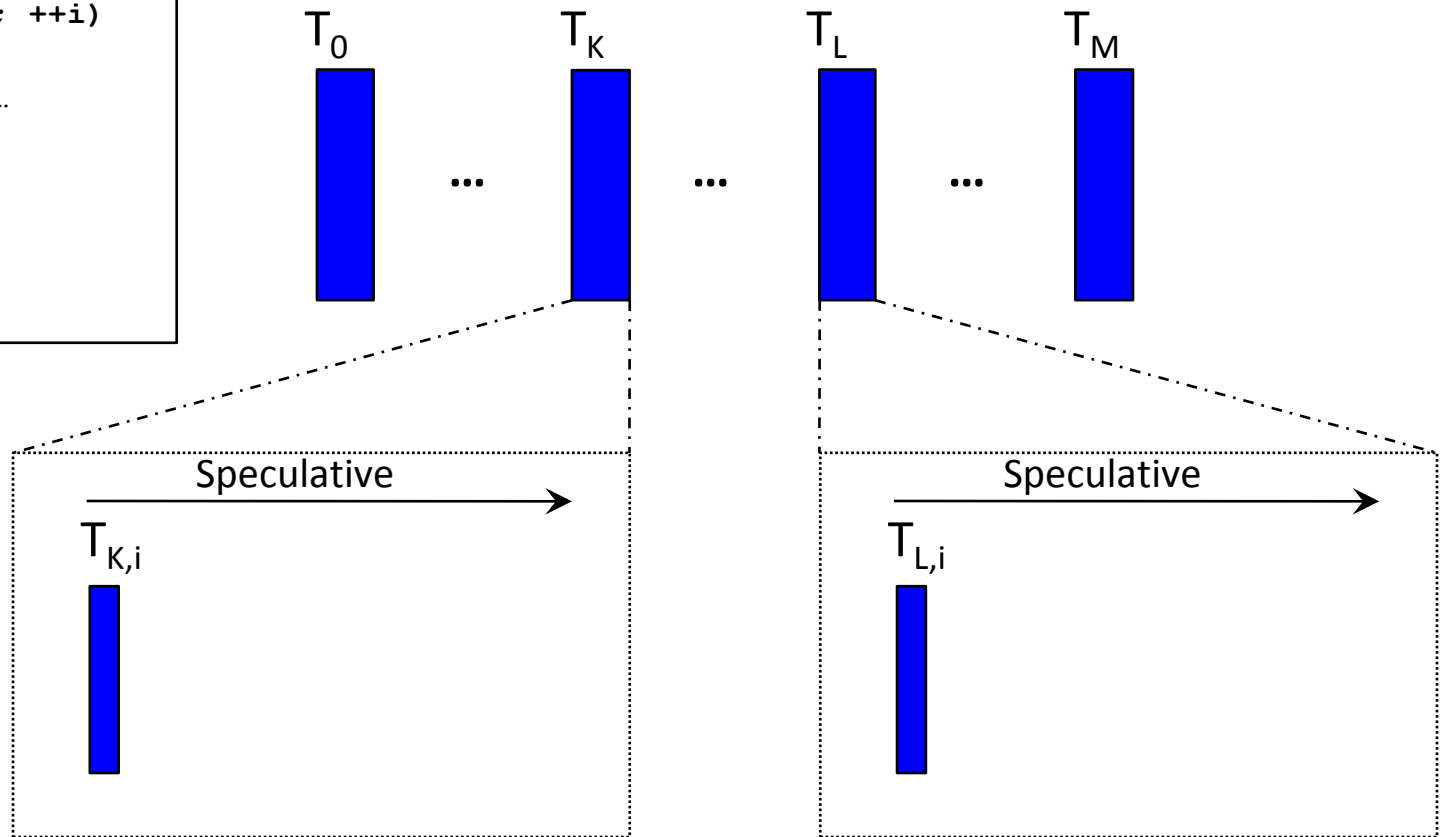
Proposal

```
#pragma omp parallel for
for(j = 0; j < M; ++j) {
  ...
  for(i = 0; i < N; ++i)
  {
    ... = A[L[i]] + ...
    ...
    A[K[i]] = ...
  }
  ...
}
```



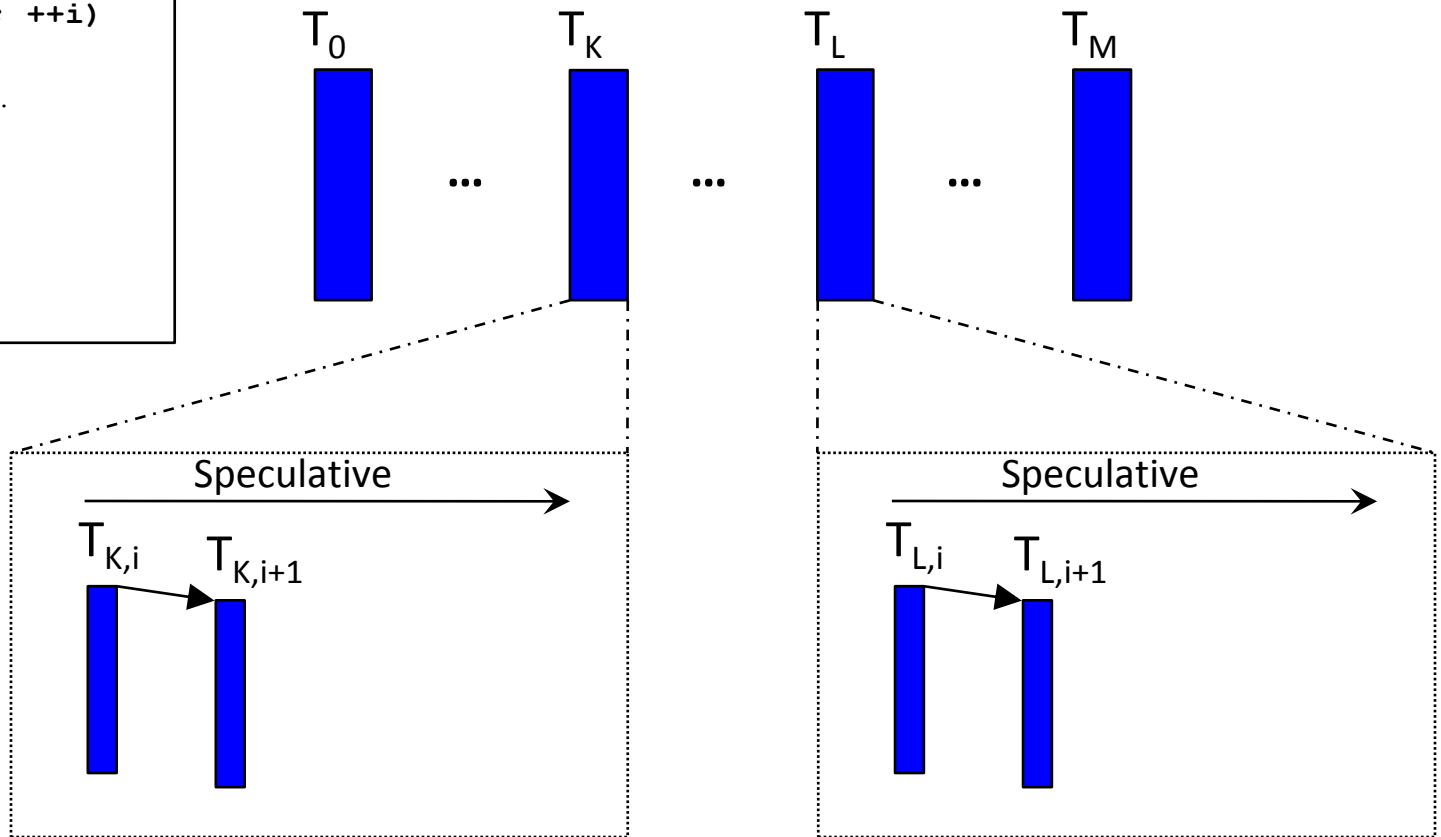
Proposal

```
#pragma omp parallel for
for(j = 0; j < M; ++j) {
  ...
  for(i = 0; i < N; ++i)
  {
    ... = A[L[i]] + ...
    ...
    A[K[i]] = ...
  }
  ...
}
```



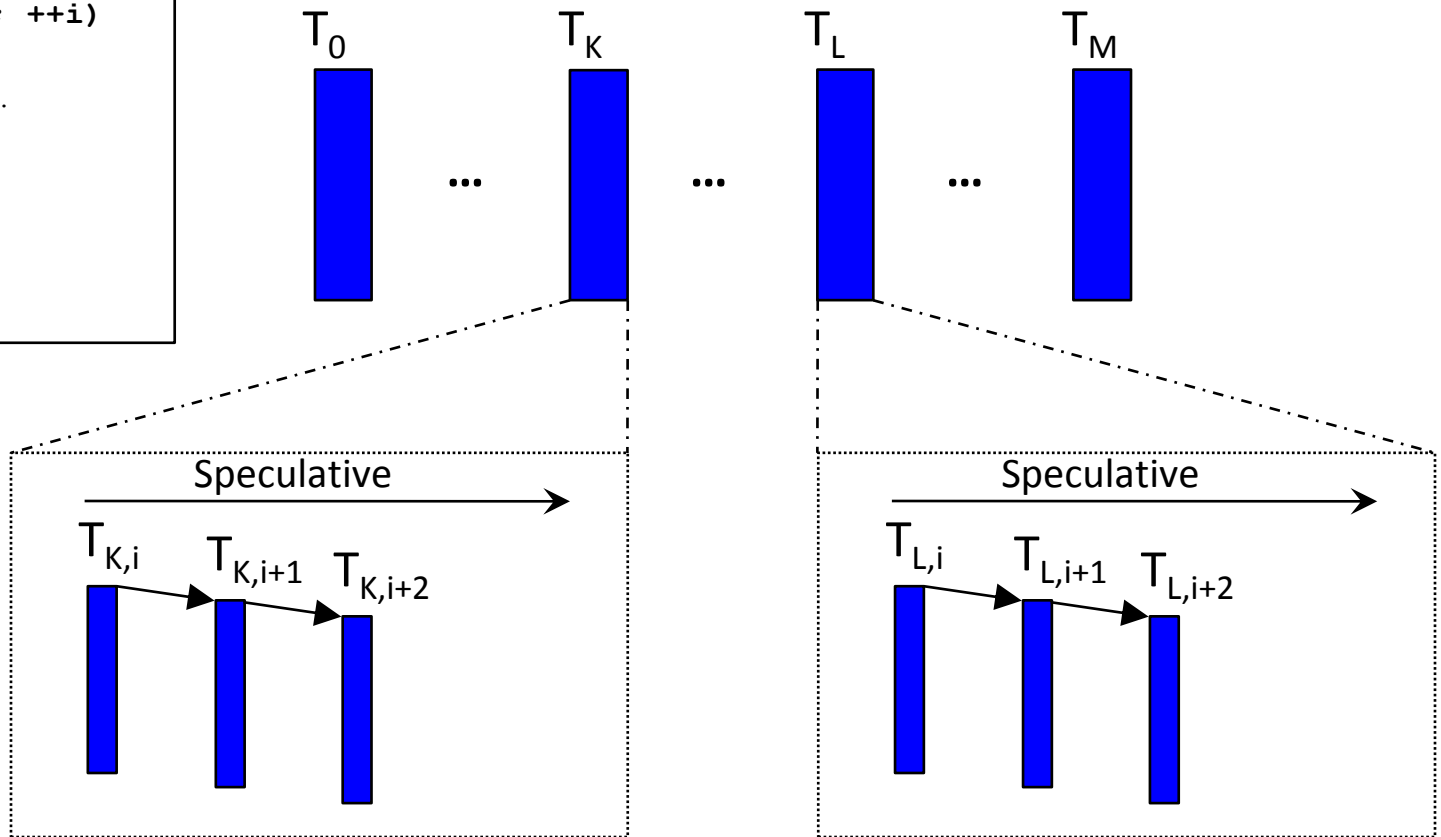
Proposal

```
#pragma omp parallel for
for(j = 0; j < M; ++j) {
  ...
  for(i = 0; i < N; ++i)
  {
    ... = A[L[i]] + ...
    ...
    A[K[i]] = ...
  }
  ...
}
```



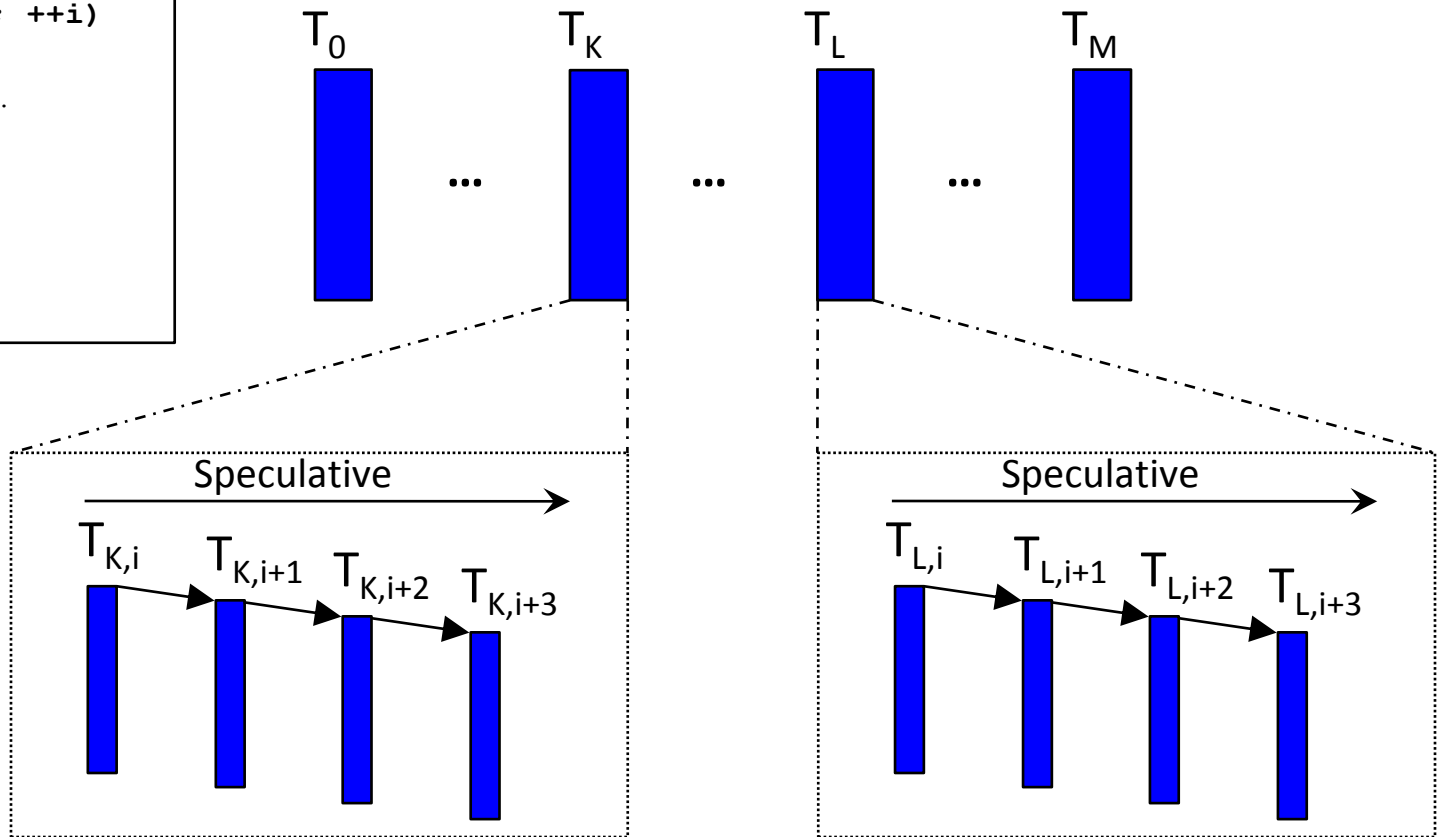
Proposal

```
#pragma omp parallel for
for(j = 0; j < M; ++j) {
  ...
  for(i = 0; i < N; ++i)
  {
    ... = A[L[i]] + ...
    ...
    A[K[i]] = ...
  }
  ...
}
```



Proposal

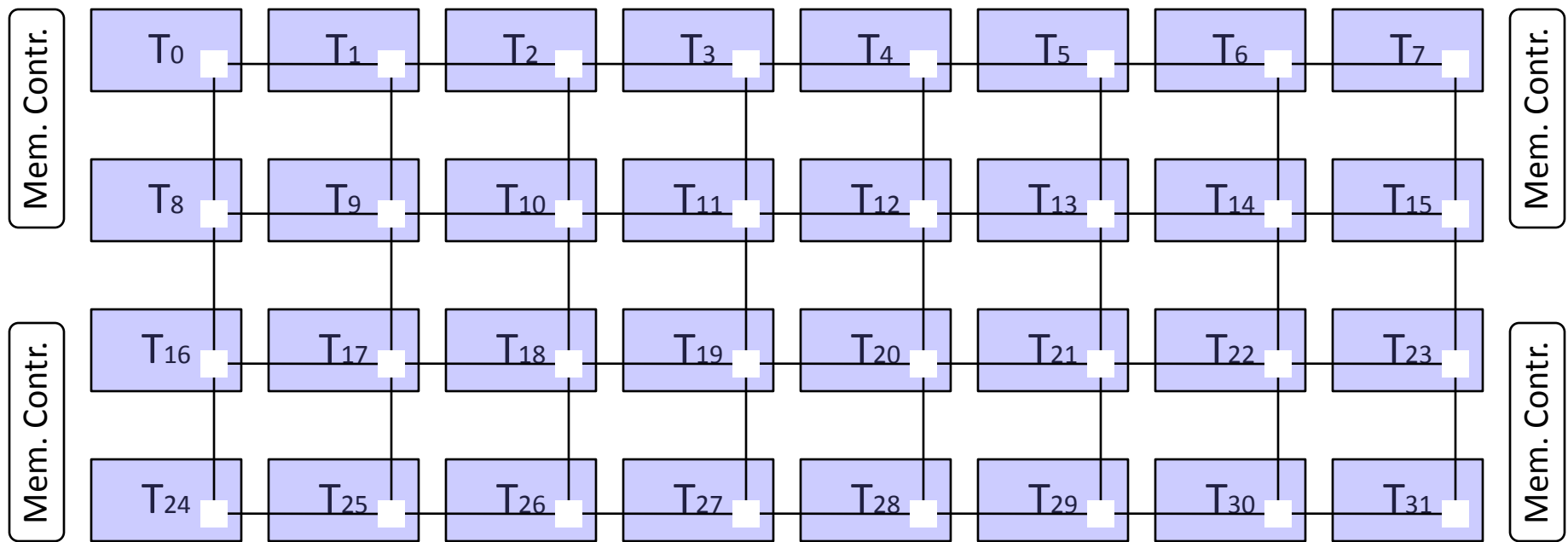
```
#pragma omp parallel for
for(j = 0; j < M; ++j) {
  ...
  for(i = 0; i < N; ++i)
  {
    ... = A[L[i]] + ...
    ...
    A[K[i]] = ...
  }
  ...
}
```



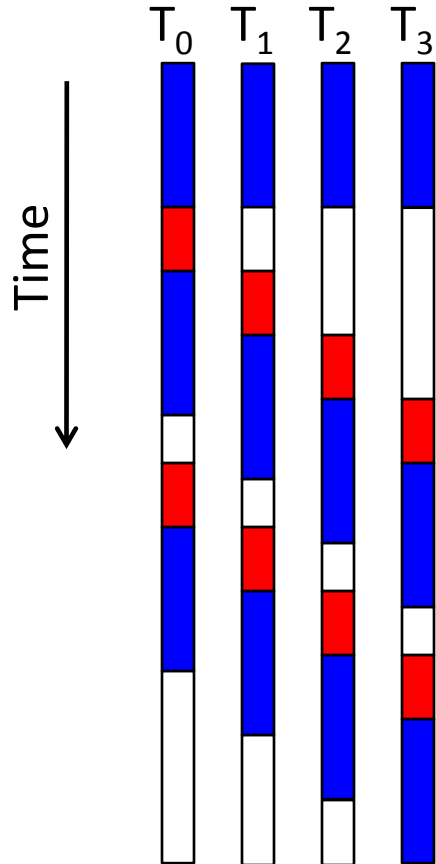
Proposal: Many-core Architecture

- Many-core partitioned in clusters (tiles)
- Coherence (MESI)
 - Snooping coherence within cluster
 - Directory coherence across clusters
- Support for TLS only within cluster
 - Snooping TLS protocol
 - Speculative buffering in L1 data caches

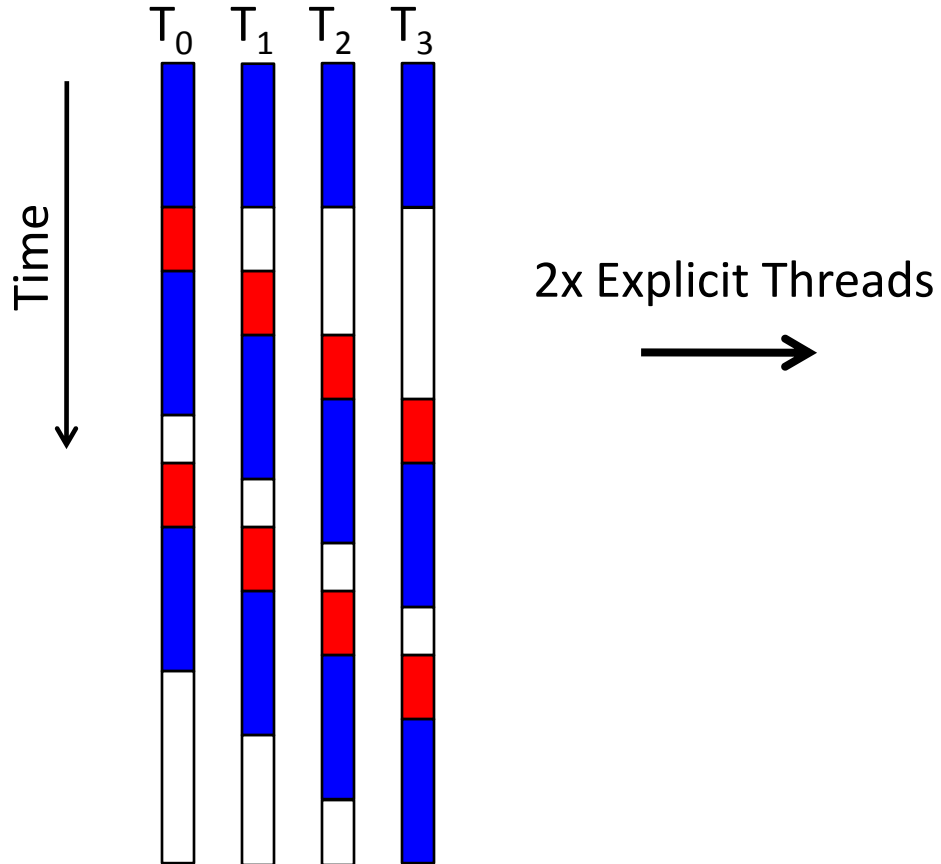
Proposal: Many-core Architecture



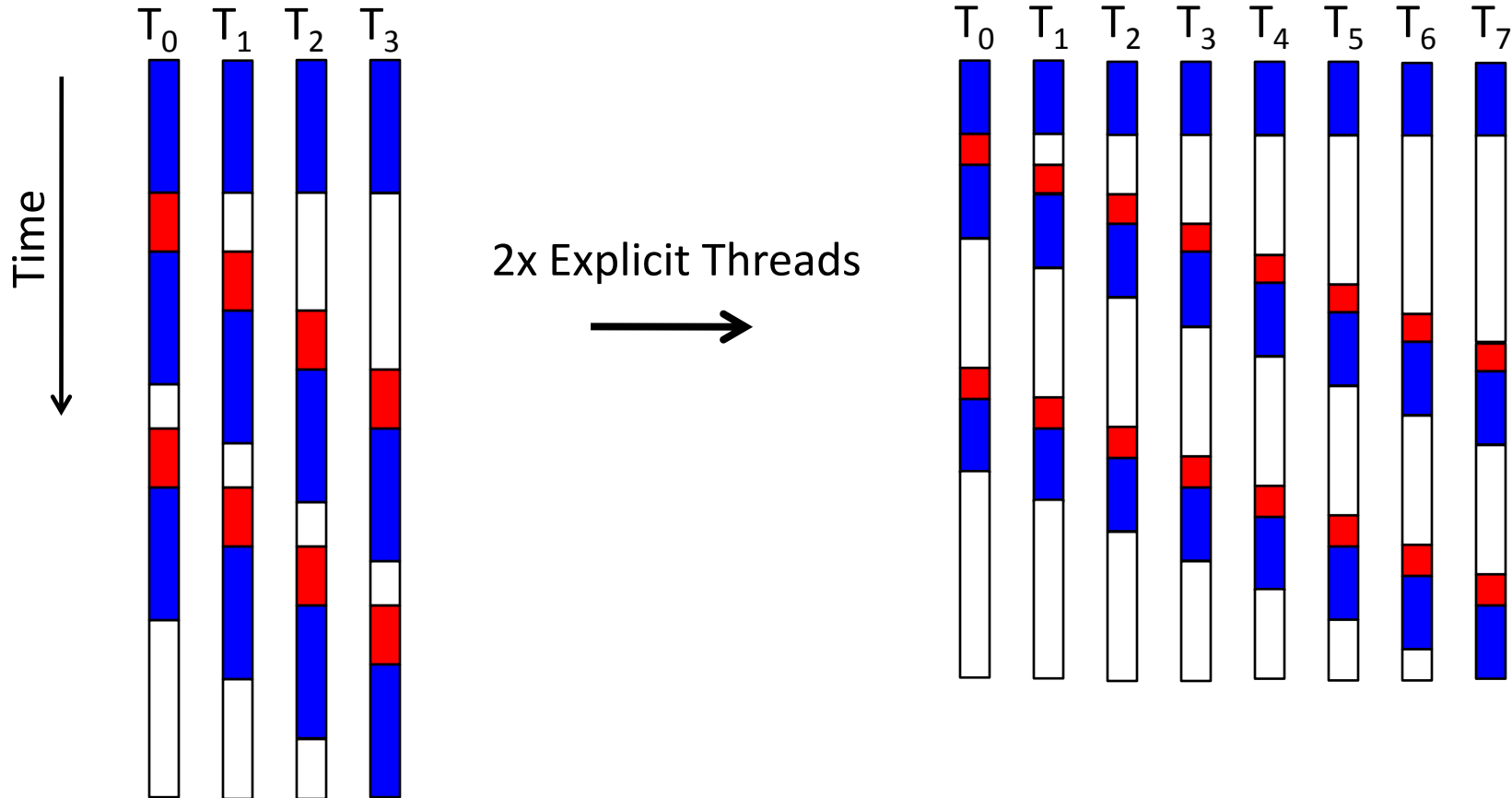
Complementing Coarse-Grain Parallelism



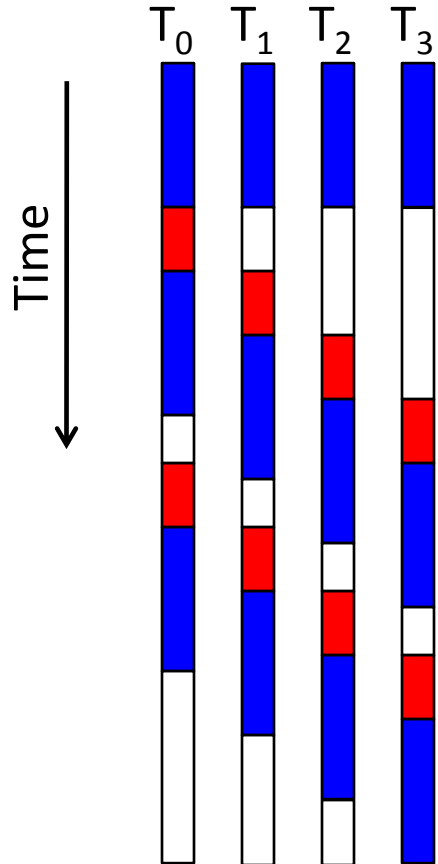
Complementing Coarse-Grain Parallelism



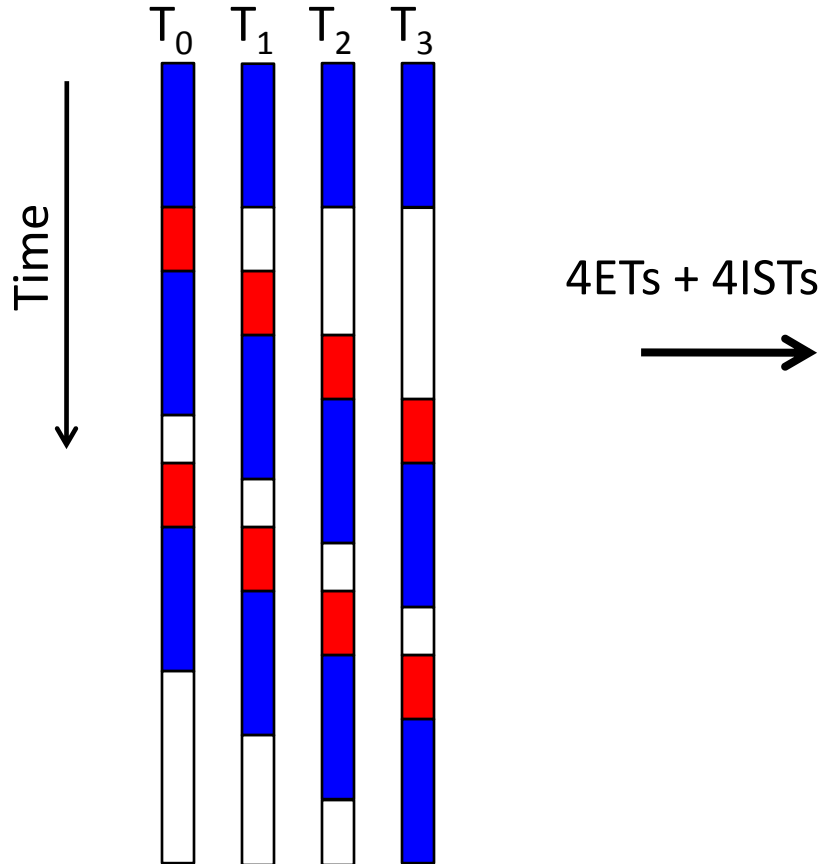
Complementing Coarse-Grain Parallelism



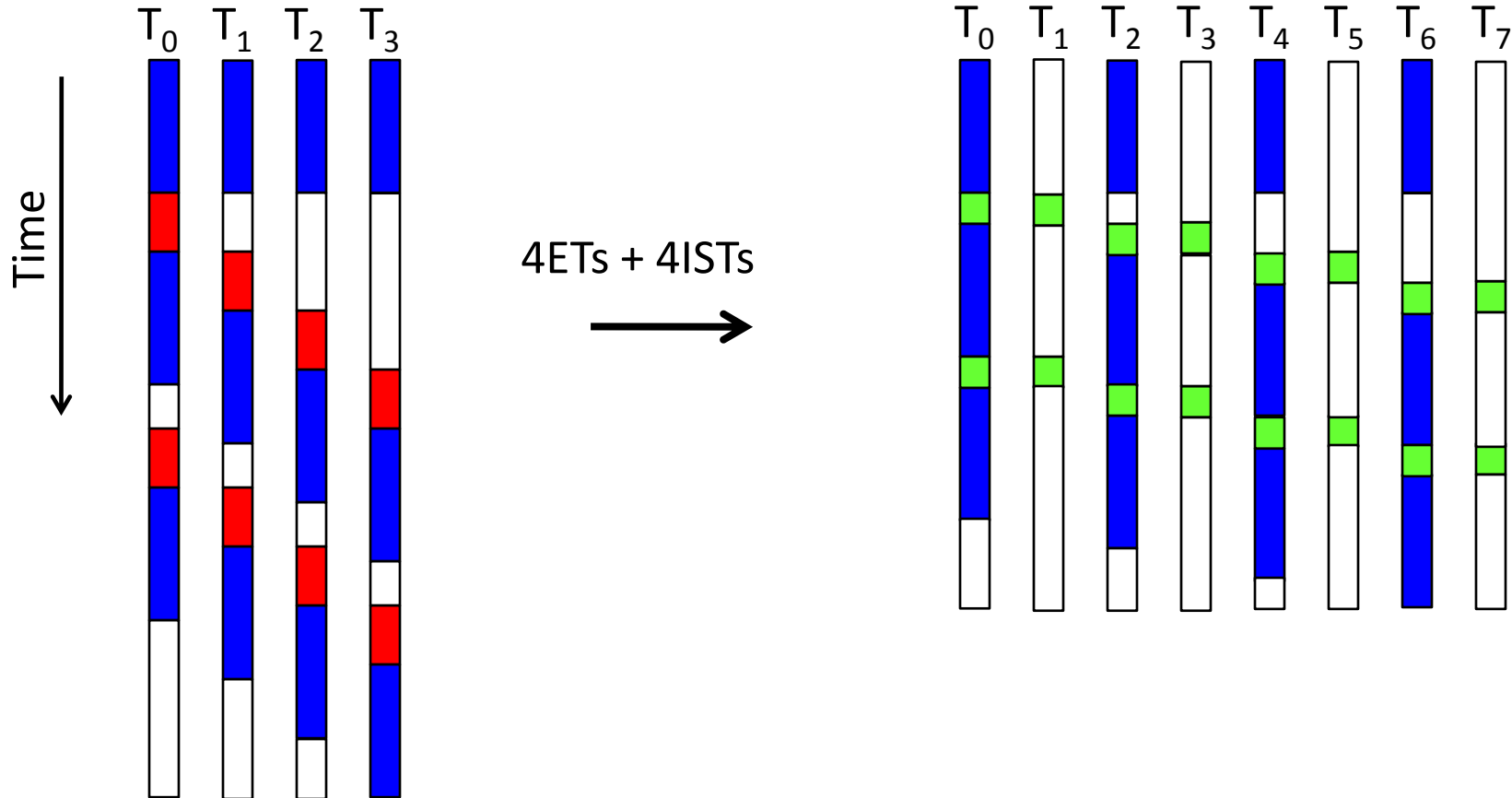
Complementing Coarse-Grain Parallelism



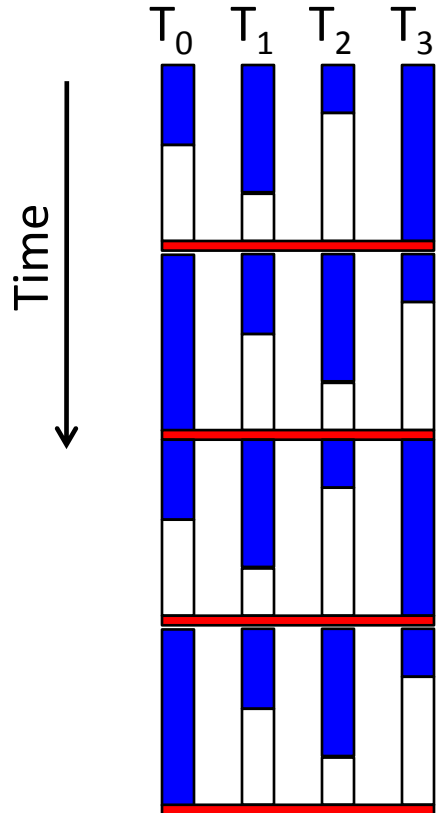
Complementing Coarse-Grain Parallelism



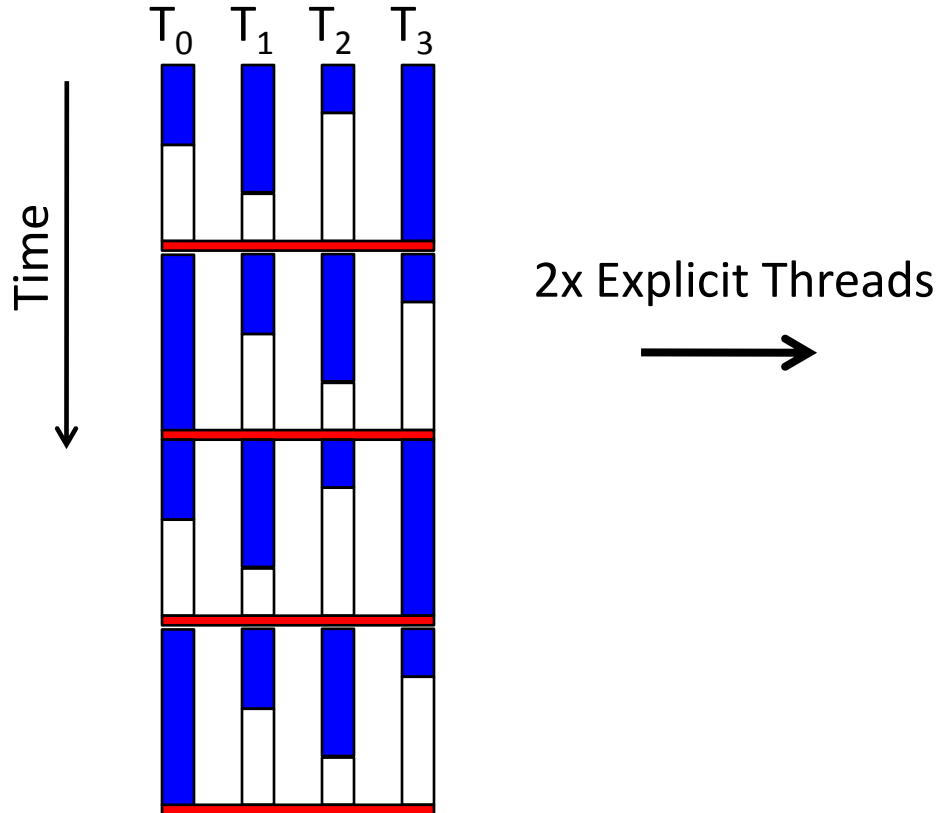
Complementing Coarse-Grain Parallelism



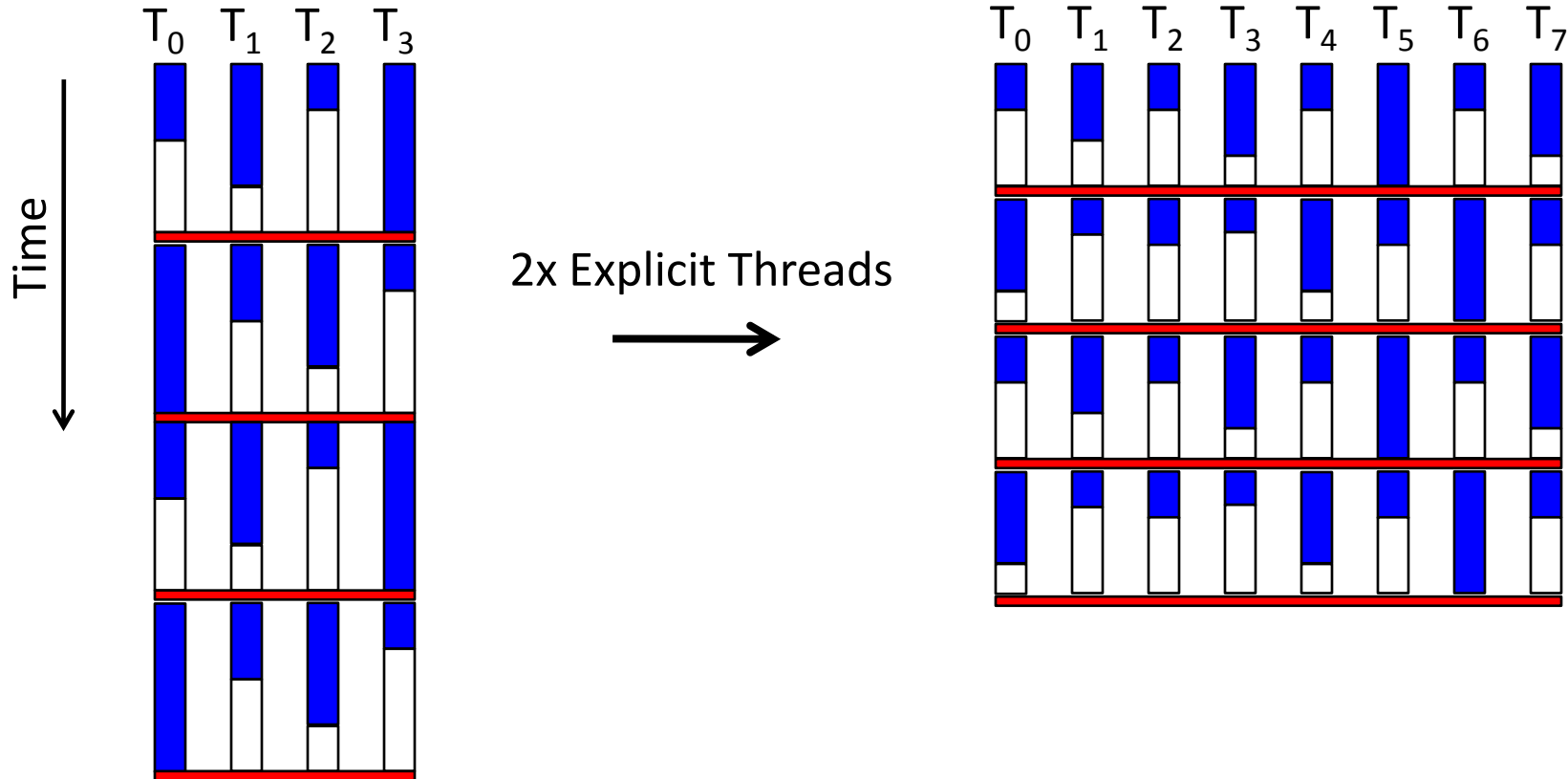
Complementing Coarse-Grain Parallelism



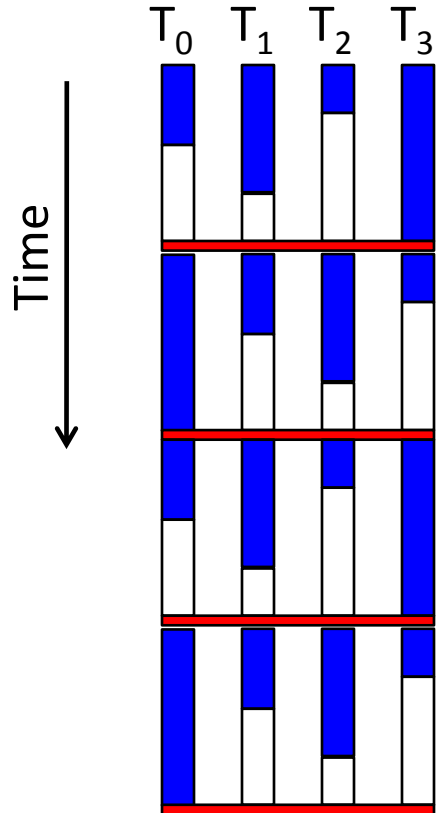
Complementing Coarse-Grain Parallelism



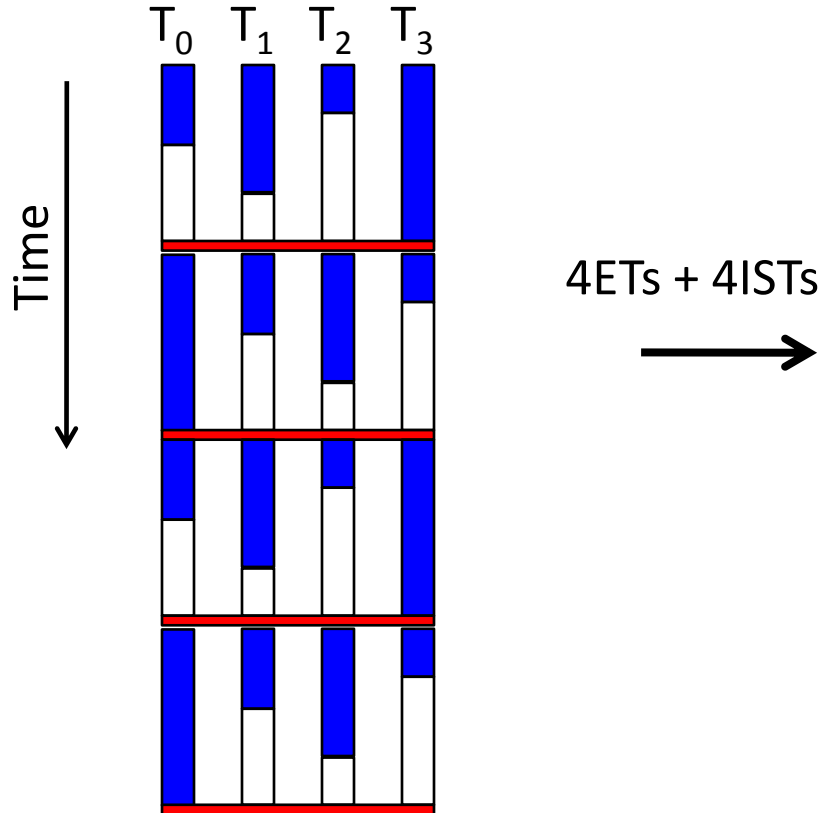
Complementing Coarse-Grain Parallelism



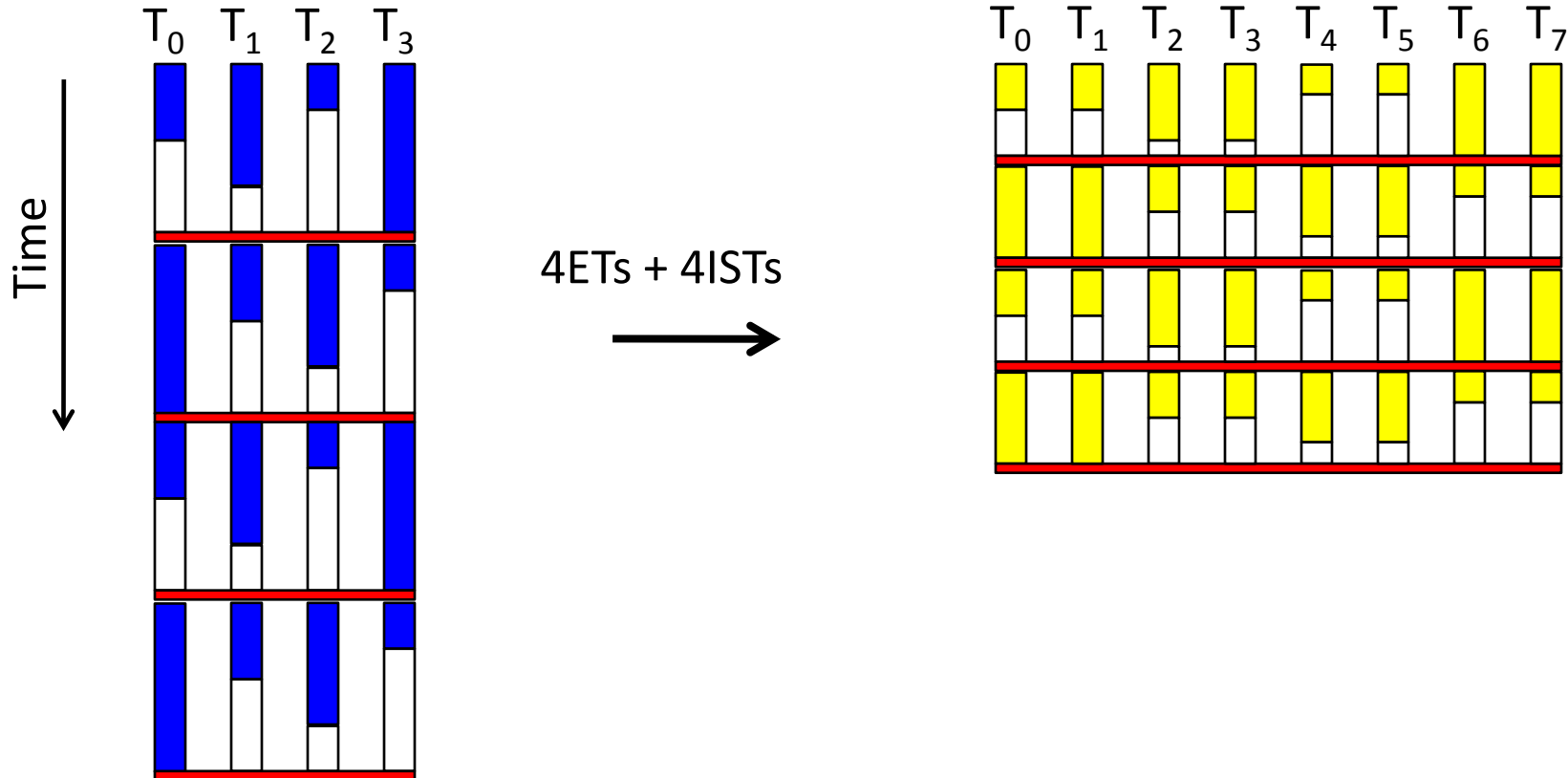
Complementing Coarse-Grain Parallelism



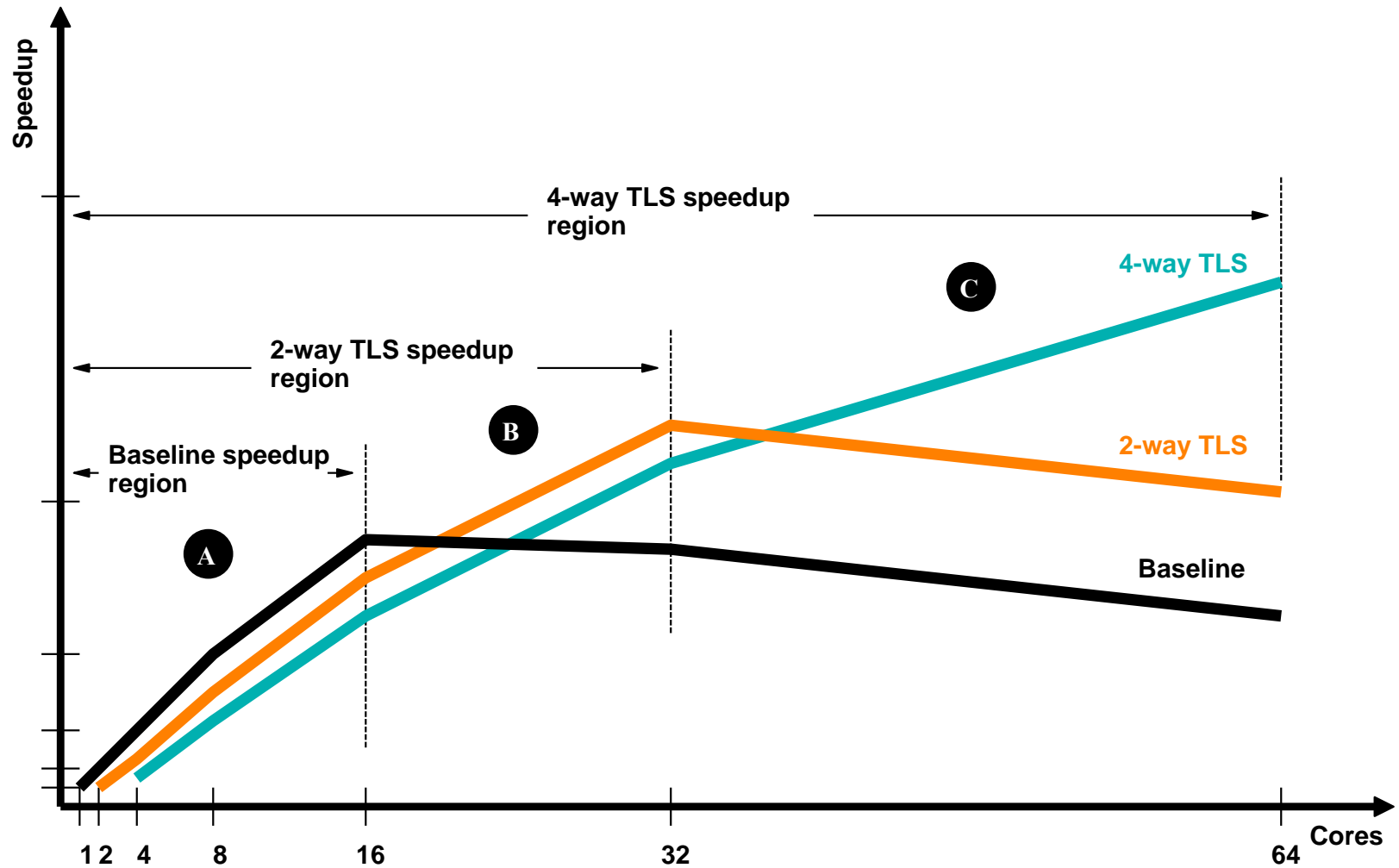
Complementing Coarse-Grain Parallelism



Complementing Coarse-Grain Parallelism



Expected Speedup Behavior



Proposal:

Auto-Tuning the Thread Count

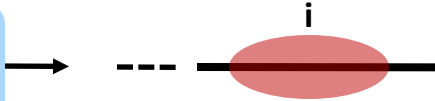
- Find the scalability tipping point dynamically
- Choose whether to employ implicit threads
- Simple hill climbing approach
- Applicable to OpenMP applications that are amenable to Dynamic Concurrency Throttling (DCT [Curtis-Maury PACT'08])
- Developed a prototype in the Omni OpenMP System

Auto-tuning example

```
...
#pragma omp parallel for
for(j = 0; j < M; ++j) {
    ...
    for(i = 0; i < N; ++i)
    {
        ... = A[L[i]] + ...
        ...
        A[K[i]] = ...
    }
    ...
}
...
```

Auto-tuning example

Learning



```
...
#pragma omp parallel for
for(j = 0; j < M; ++j) {
    ...
    for(i = 0; i < N; ++i)
    {
        ... = A[L[i]] + ...
        ...
        A[K[i]] = ...
    }
    ...
}
...
```

Auto-tuning example

Learning →



omp parallel region *i* detected:

First time:

Can we compute iteration count
statically and is less than max
core count?

```
...  
#pragma omp parallel for  
for(j = 0; j < M; ++j) {  
  ...  
  for(i = 0; i < N; ++i)  
  {  
    ... = A[L[i]] + ...  
    ...  
    A[K[i]] = ...  
  }  
  ...  
}  
...
```

Auto-tuning example

Learning



omp parallel region *i* detected:

First time:

Can we compute iteration count
statically and is less than max
core count?

```
...  
#pragma omp parallel for  
for(j = 0; j < M; ++j) {  
...  
  for(i = 0; i < N; ++i)  
  {  
    ... = A[L[i]] + ...  
    ...  
    A[K[i]] = ...  
  }  
...  
}  
...
```

M=32

Auto-tuning example

Learning →



```
...  
#pragma omp parallel for  
for(j = 0; j < M; ++j) {  
...  
  for(i = 0; i < N; ++i)  
  {  
    ... = A[L[i]] + ...  
    ...  
    A[K[i]] = ...  
  }  
...  
}
```

M=32

omp parallel region *i* detected:

First time:

Can we compute iteration count
statically and is less than max
core count?

Yes -> set Initial *Tcount* to 32

Auto-tuning example

Learning



omp parallel region i detected:

First time:

Can we compute iteration count statically and is less than max core count?

Yes -> set Initial $Tcount$ to 32

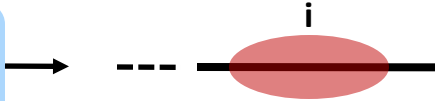
Measure execution time t_i^1

```
...
#pragma omp parallel for
for(j = 0; j < M; ++j) {
  ...
  for(i = 0; i < N; ++i)
  {
    ... = A[L[i]] + ...
    ...
    A[K[i]] = ...
  }
  ...
}
...
```

M=32

Auto-tuning example

Learning



```
...  
#pragma omp parallel for  
for(j = 0; j < M; ++j) {  
    ...  
    for(i = 0; i < N; ++i)  
    {  
        ... = A[L[i]] + ...  
        ...  
        A[K[i]] = ...  
    }  
    ...  
}  
...
```

Auto-tuning example

Learning →



```
...  
#pragma omp parallel for  
for(j = 0; j < M; ++j) {  
  ...  
  for(i = 0; i < N; ++i)  
  {  
    ... = A[L[i]] + ...  
    ...  
    A[K[i]] = ...  
  }  
  ...  
}  
...
```

Auto-tuning example

Learning →



```
...  
#pragma omp parallel for  
for(j = 0; j < M; ++j) {  
  ...  
  for(i = 0; i < N; ++i)  
  {  
    ... = A[L[i]] + ...  
    ...  
    A[K[i]] = ...  
  }  
  ...  
}  
...
```

omp parallel region *i* detected:

Set *Tcount* to next value (16)

Measure execution time t_i^2

$t_i^2 < t_i^1 \rightarrow$ continue exploration

Auto-tuning example

Learning →



```
...  
#pragma omp parallel for  
for(j = 0; j < M; ++j) {  
  ...  
  for(i = 0; i < N; ++i)  
  {  
    ... = A[L[i]] + ...  
    ...  
    A[K[i]] = ...  
  }  
  ...  
}  
...
```

Auto-tuning example

Learning →



```
...  
#pragma omp parallel for  
for(j = 0; j < M; ++j) {  
  ...  
  for(i = 0; i < N; ++i)  
  {  
    ... = A[L[i]] + ...  
    ...  
    A[K[i]] = ...  
  }  
  ...  
}  
...
```

Auto-tuning example

Learning →



```
...  
#pragma omp parallel for  
for(j = 0; j < M; ++j) {  
  ...  
  for(i = 0; i < N; ++i)  
  {  
    ... = A[L[i]] + ...  
    ...  
    A[K[i]] = ...  
  }  
  ...  
}  
...
```

omp parallel region *i* detected:

Set *Tcount* to next value (8)

Measure execution time t_i^3

$t_i^3 > t_i^2 \rightarrow$ stop exploration

Auto-tuning example

Learning →



```
...  
#pragma omp parallel for  
for(j = 0; j < M; ++j) {  
  ...  
  for(i = 0; i < N; ++i)  
  {  
    ... = A[L[i]] + ...  
    ...  
    A[K[i]] = ...  
  }  
  ...  
}  
...
```

Auto-tuning example

Learning →



```
...  
#pragma omp parallel for  
for(j = 0; j < M; ++j) {  
  ...  
  for(i = 0; i < N; ++i)  
  {  
    ... = A[L[i]] + ...  
    ...  
    A[K[i]] = ...  
  }  
  ...  
}  
...
```

Auto-tuning example

Learning →



```
...  
#pragma omp parallel for  
for(j = 0; j < M; ++j) {  
  ...  
  for(i = 0; i < N; ++i)  
  {  
    ... = A[L[i]] + ...  
    ...  
    A[K[i]] = ...  
  }  
  ...  
}  
...
```

omp parallel region *i* detected:
Use Tcount = 16, no further
exploration
Set TLS to 4-way

Outline

- Introduction
- Motivation
- Proposal
- Evaluation Methodology
- Results
- Conclusions

Evaluation Methodology

- SESC simulator - extended to model our scheme
- Architecture:
 - Core:
 - 4-issue OoO superscalar, 96-entry ROB, 3GHz
 - 32KB, 4-way, DL1 \$ - 32KB, 2-way, IL1 \$
 - 16Kbit Hybrid Branch Predictor
 - Tile/System:
 - 128 cores partitioned in 2-way or 4-way tiles (evaluate both)
 - Shared L2 cache, 8MB, 8-way, 64MSHRs
 - Directory: Full-bit vector sharer list
 - Interconnect: Grid, 64B links - 48GB/s to main memory

Evaluation Methodology

- Benchmarks:
 - 12 workloads from PARSEC 2.1, SPLASH2, NASPB
 - Simulate parallel region to completion
- Compilation:
 - MIPS binaries generated using GCC 3.4.4
 - Speculation added automatically through source-to-source compiler
 - Selection of speculation regions through manual profiling
- Power:
 - CACTI 4.2 and Wattch

Evaluation Methodology

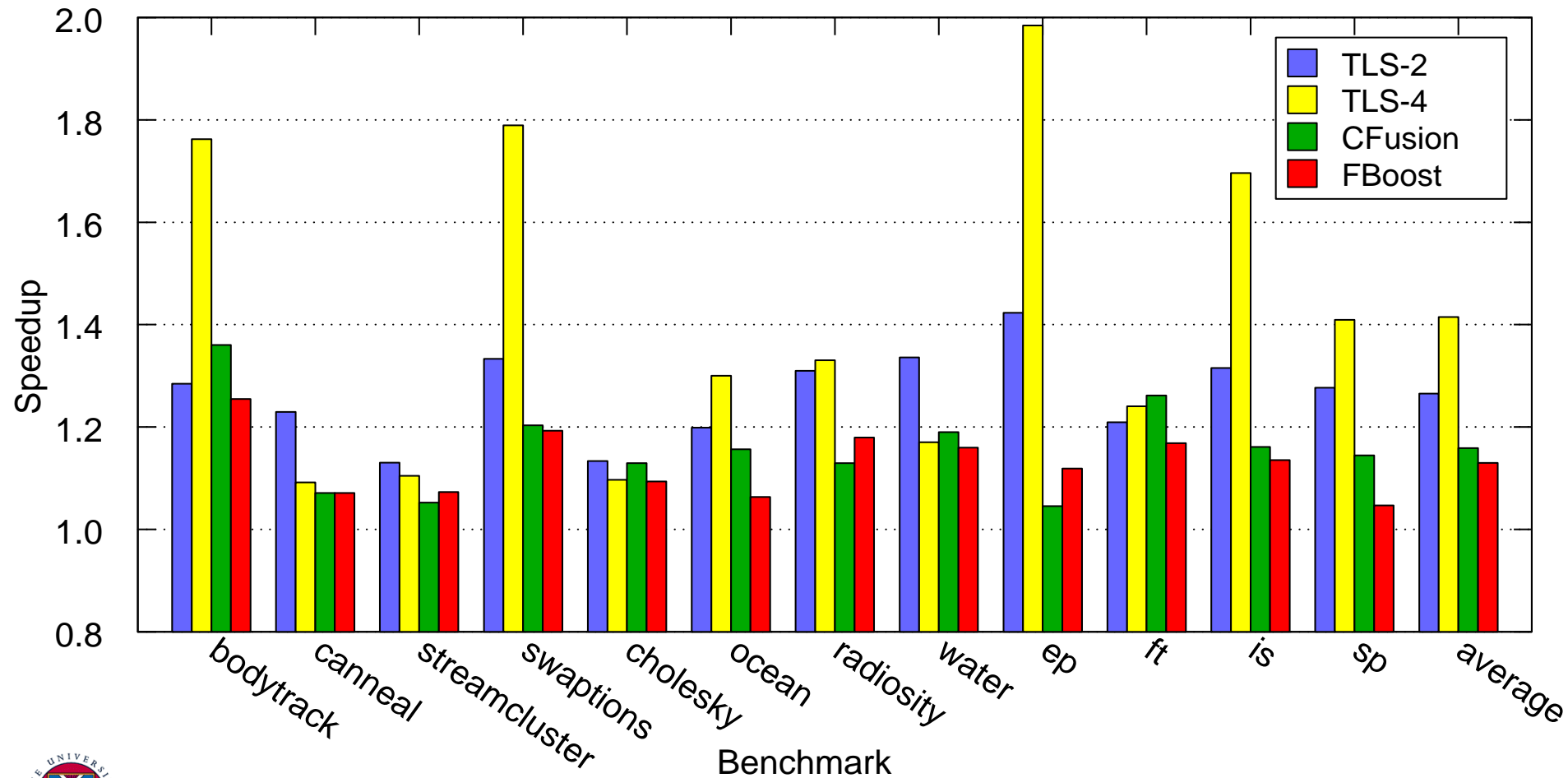
- Alternative schemes compared against:
 - Core Fusion [Ipek ISCA'07]:
 - Dynamic combination of cores to deal with lowly-threaded apps
 - Approximated through wide 8-issue cores with all the core resources doubled without latency increase => *upper bound*
 - Frequency Boost:
 - Inspired by Turbo Boost [Intel'08]
 - For each idle core one other core gains a frequency boost of 800MHz with a 200mV increase in voltage (same power cap)
- All these schemes shift resources to a subset of cores in order to improve performance

Outline

- Introduction
- Motivation
- Proposal
- Evaluation Methodology
- Results
- Conclusions

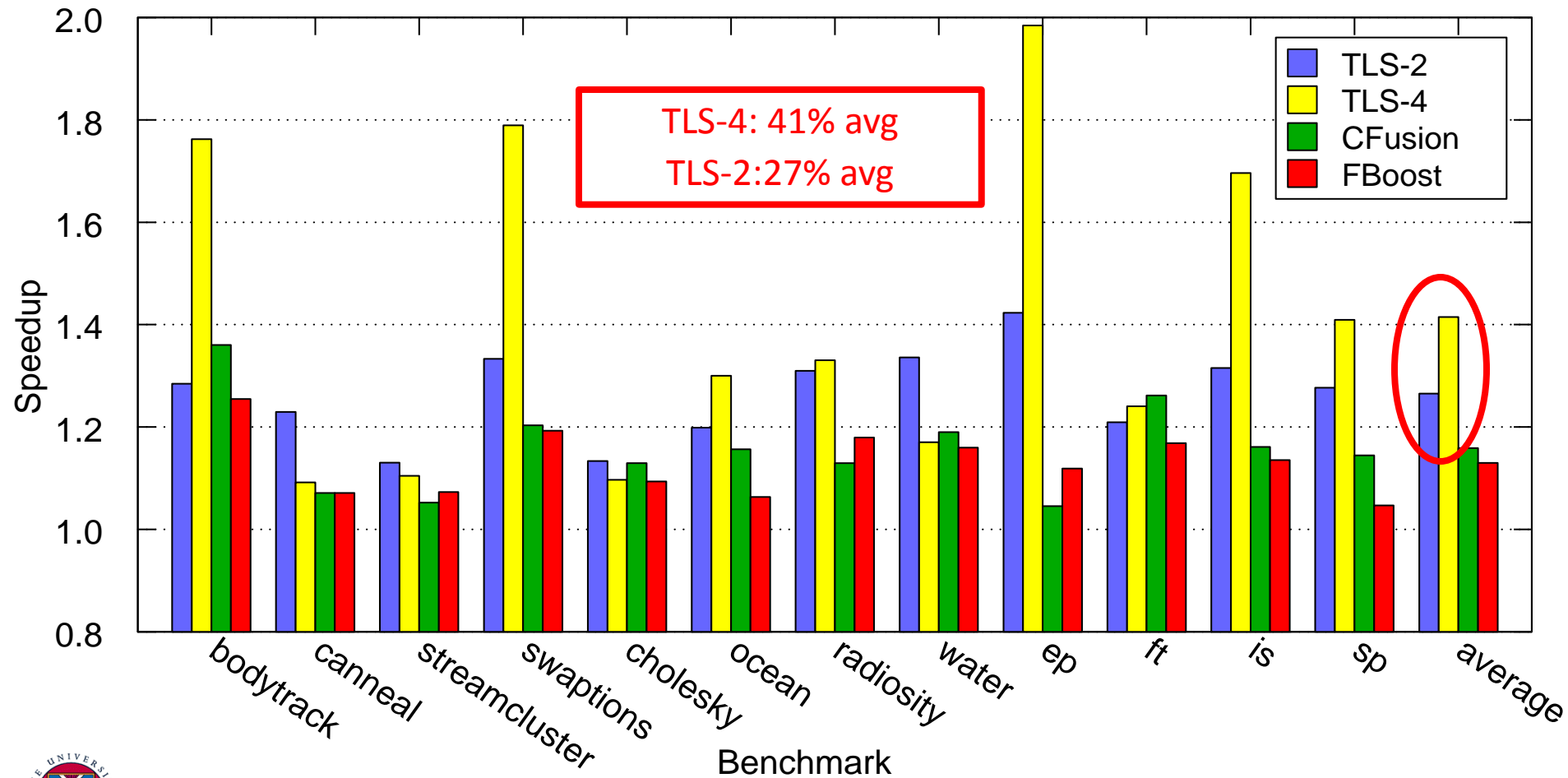
Bottom Line

- Speedup over *best* scalability point



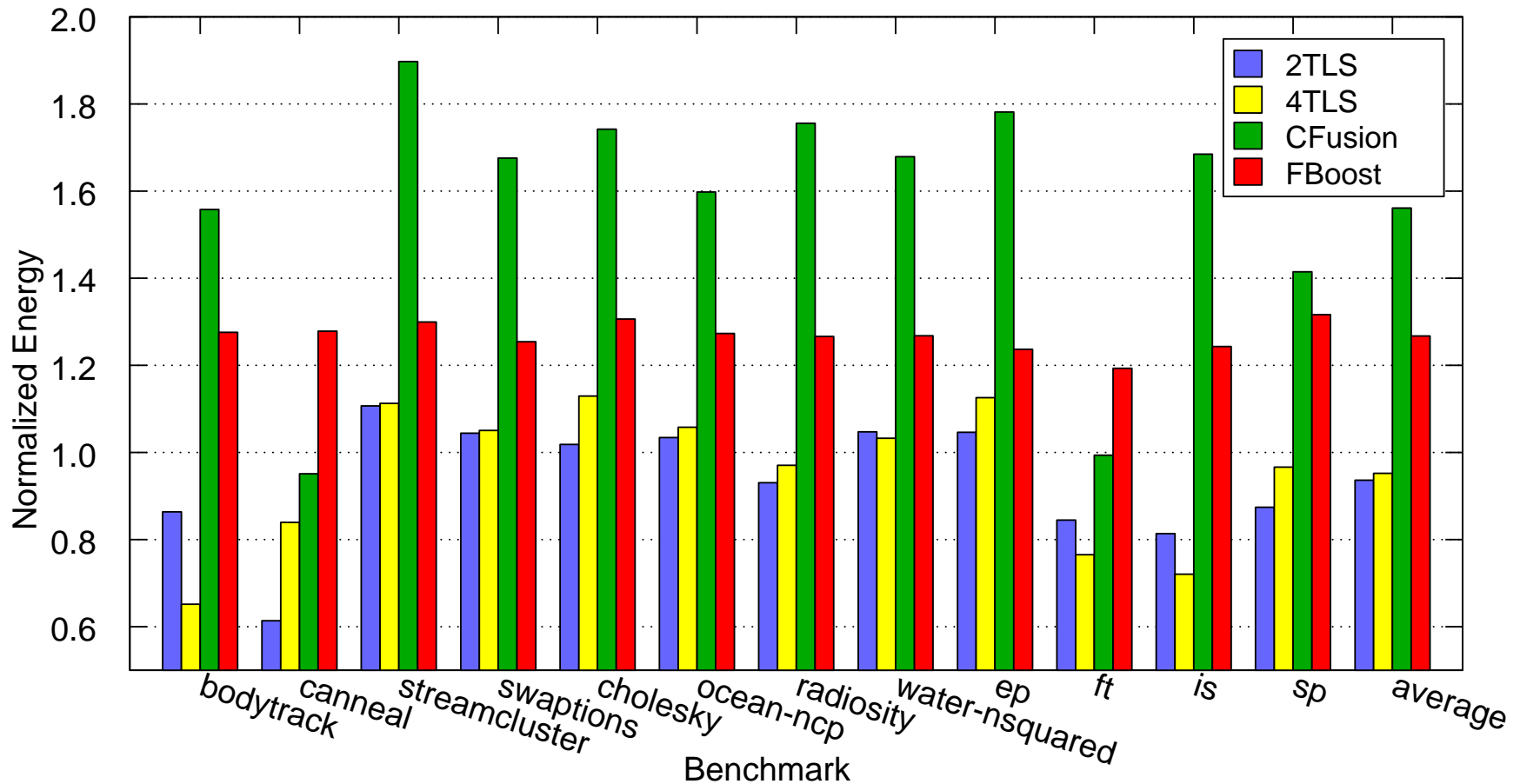
Bottom Line

- Speedup over *best* scalability point



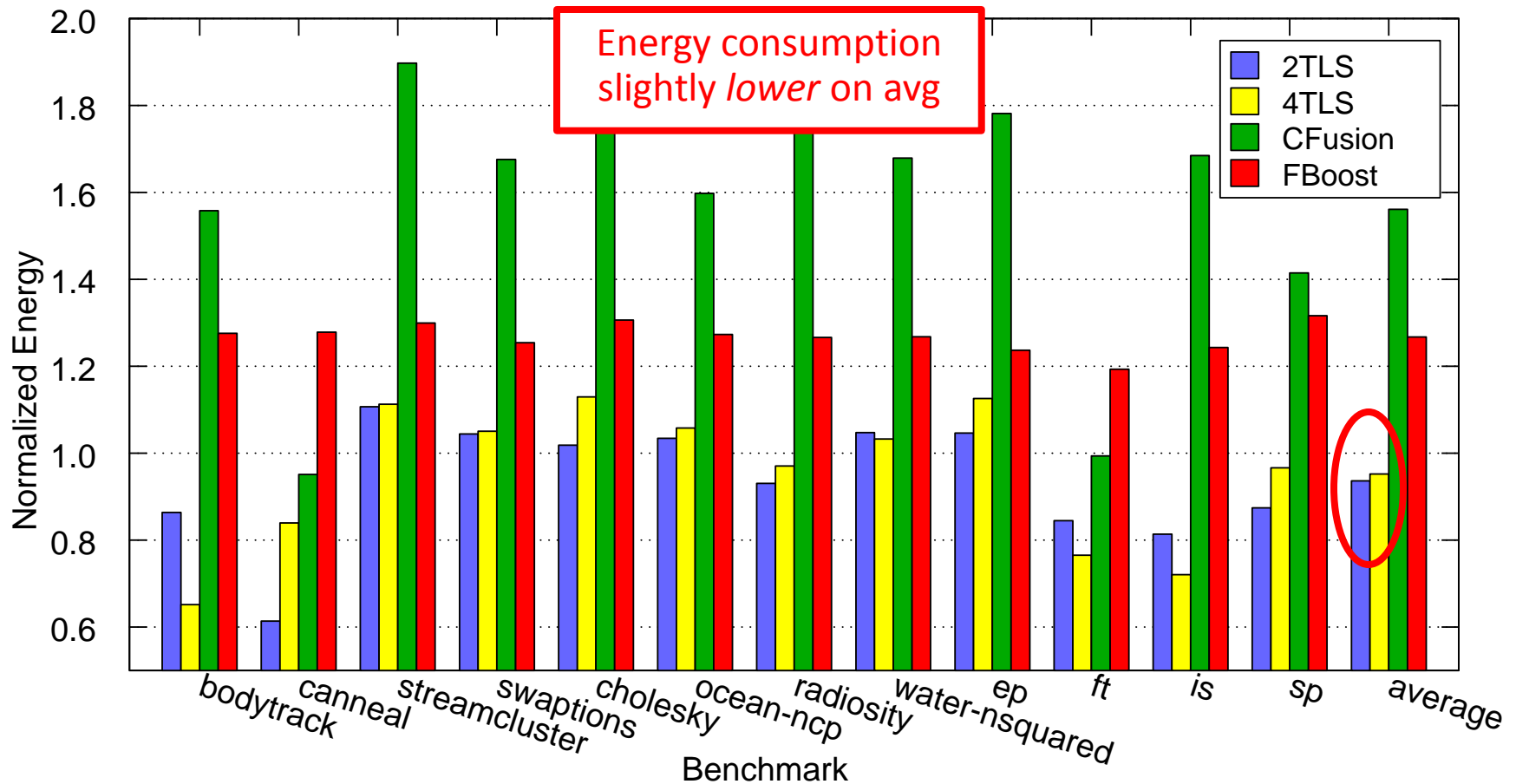
Energy

- Showing best performing point for each scheme



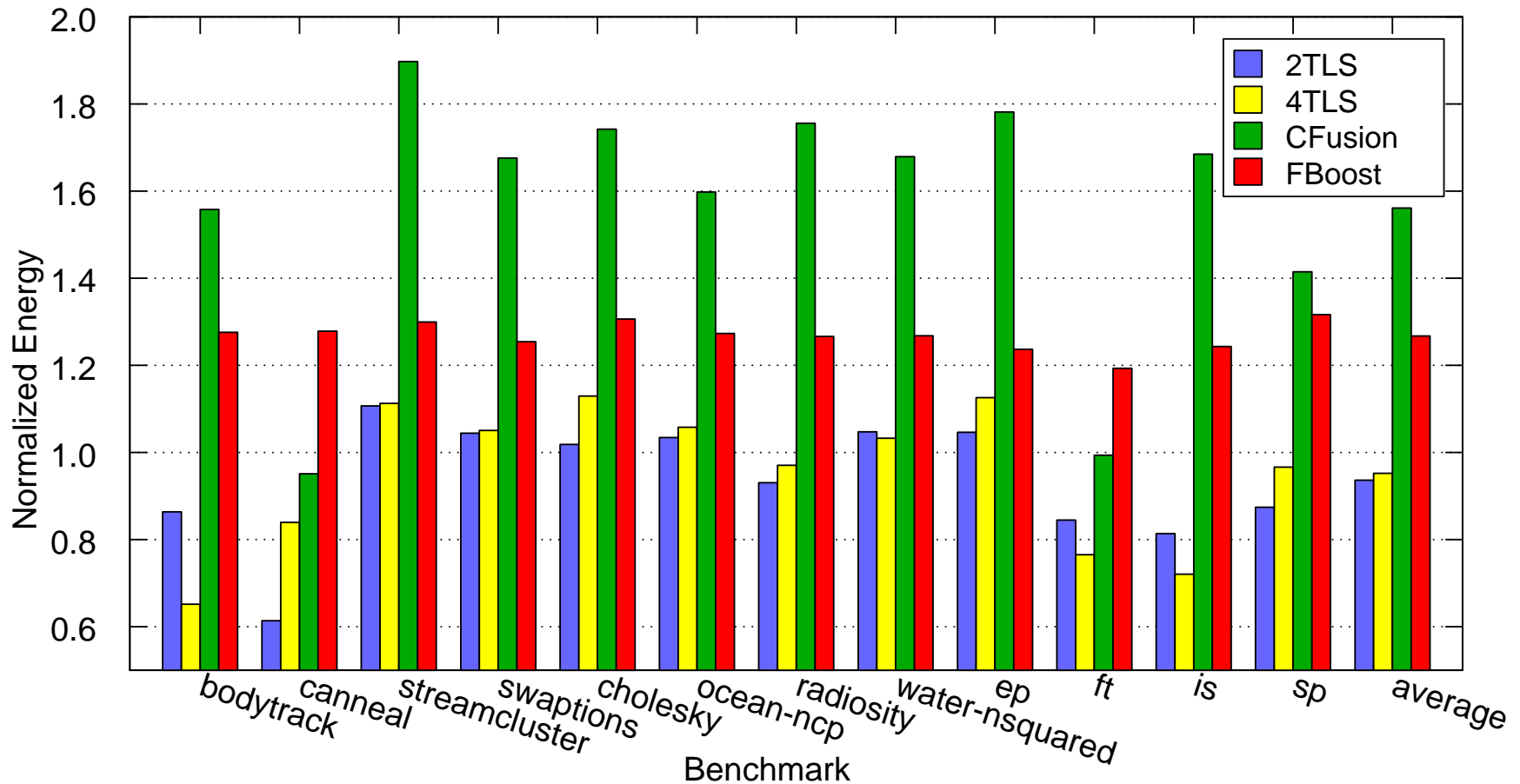
Energy

- Showing best performing point for each scheme



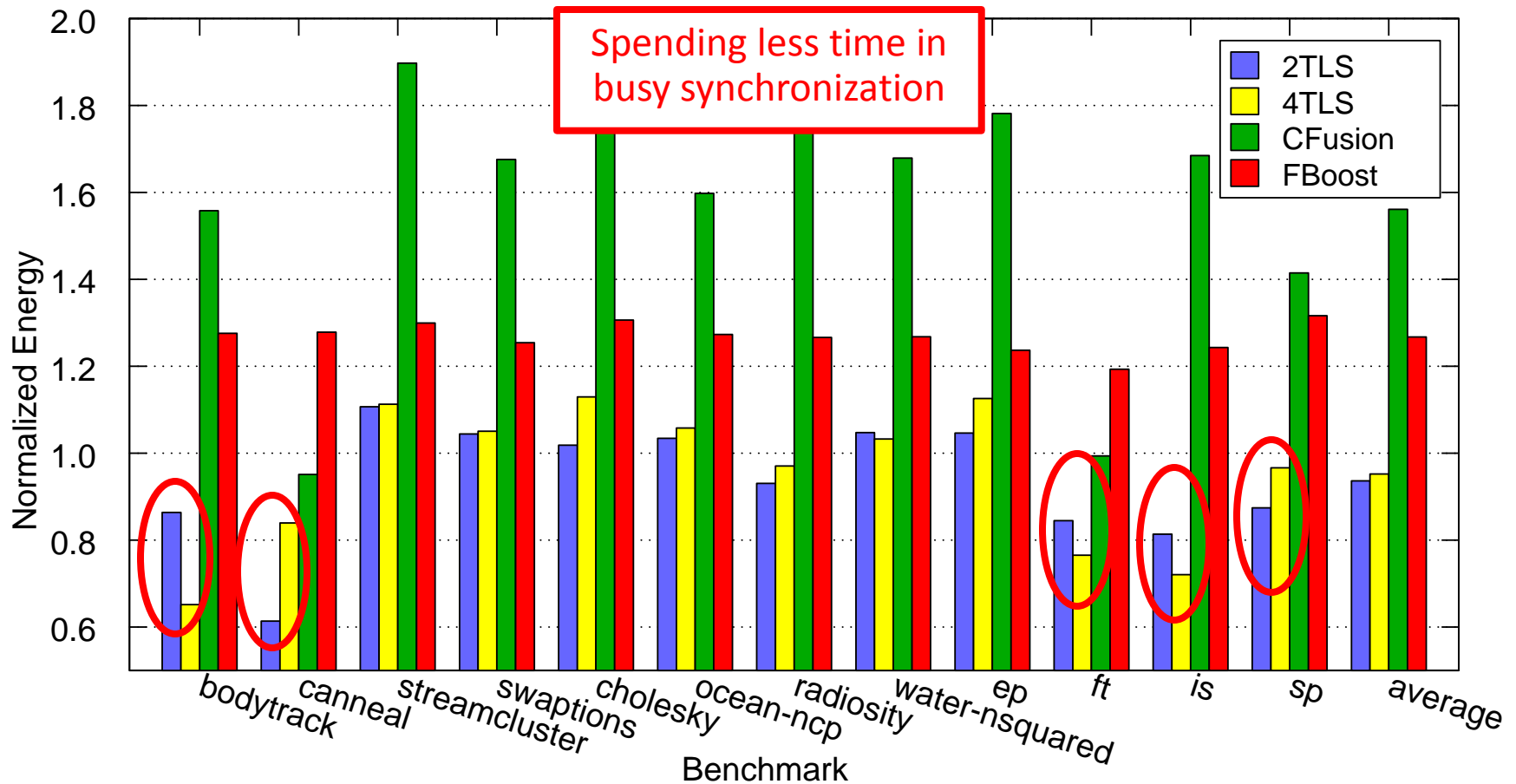
Energy

- Showing best performing point for each scheme



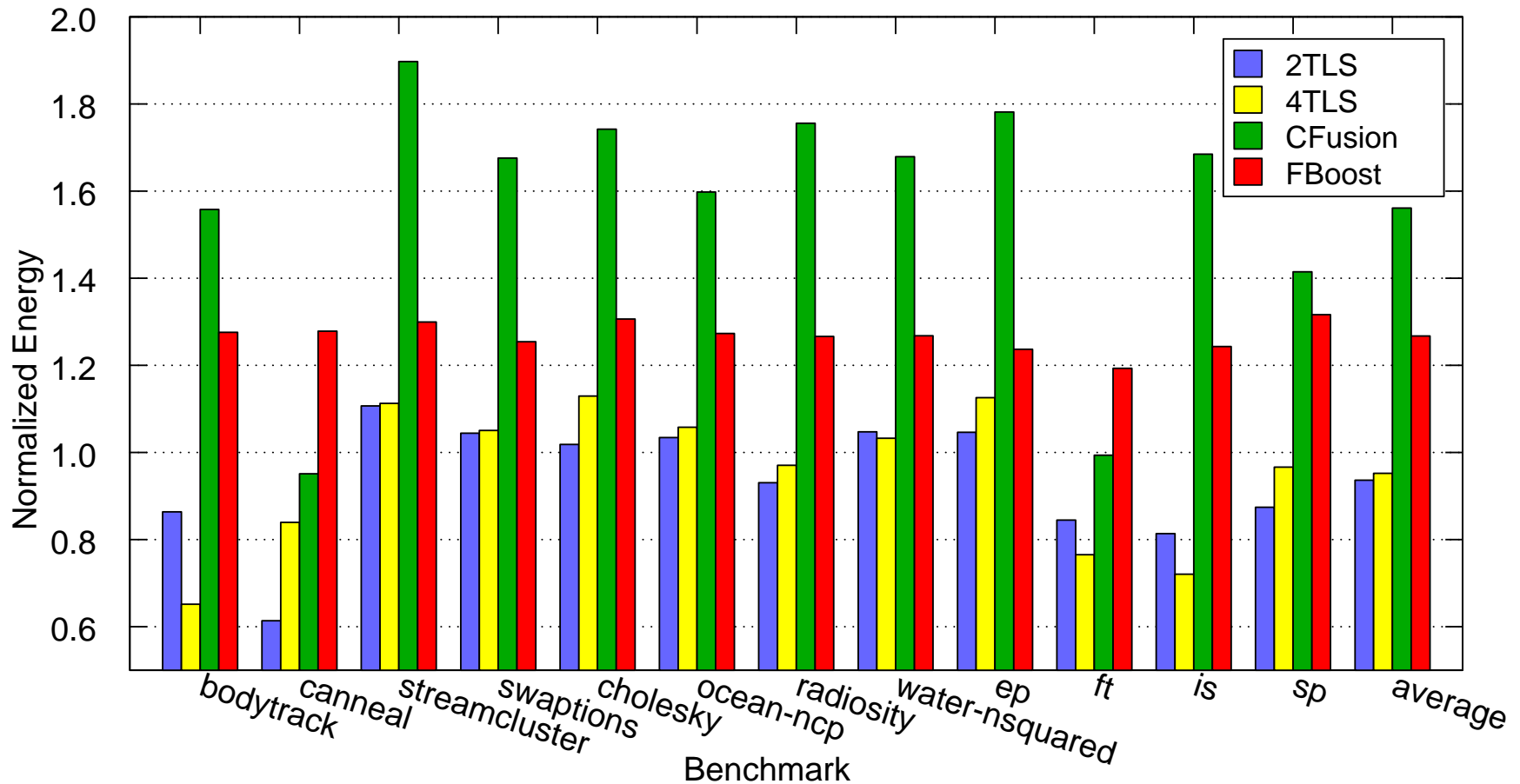
Energy

- Showing best performing point for each scheme



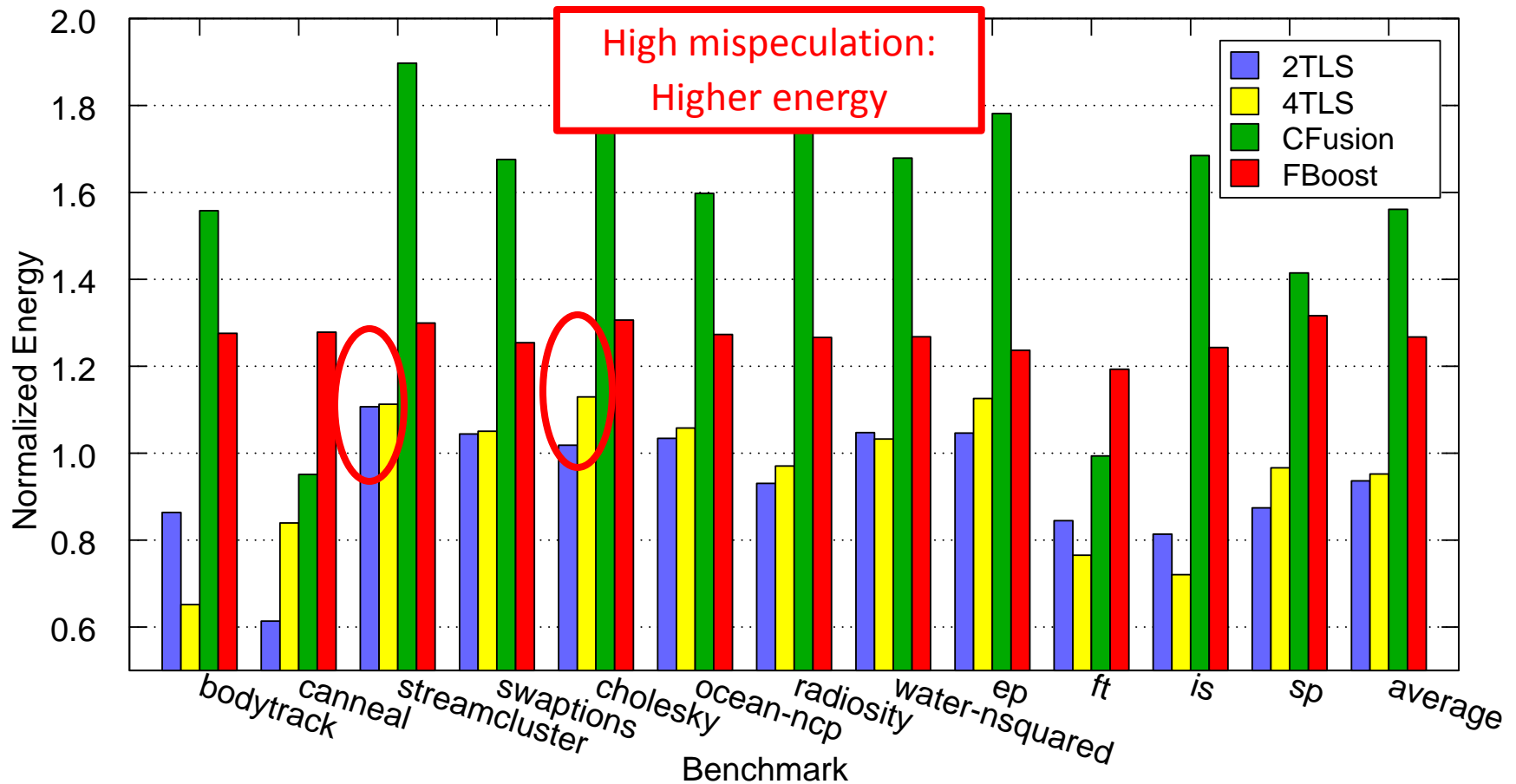
Energy

- Showing best performing point for each scheme



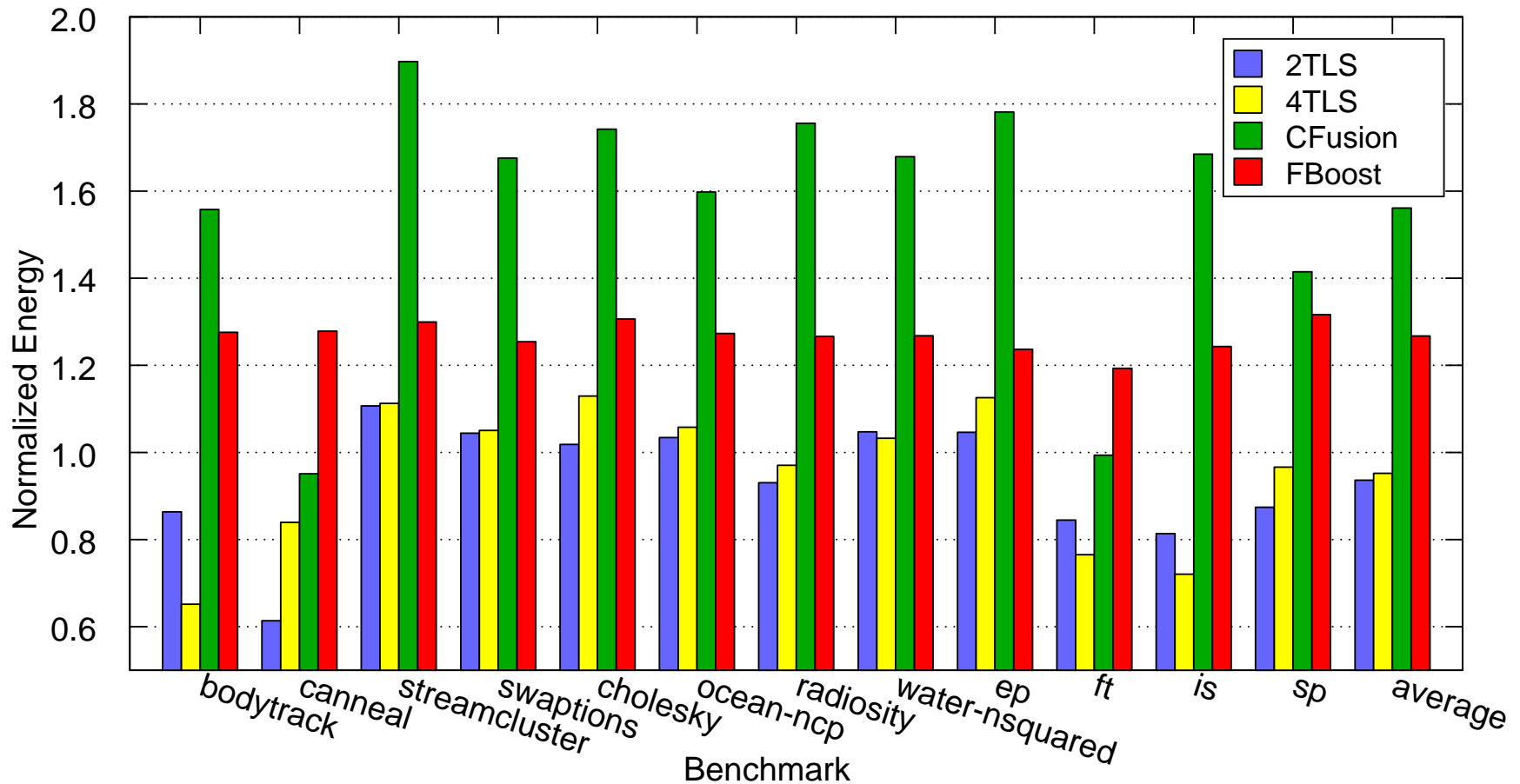
Energy

- Showing best performing point for each scheme



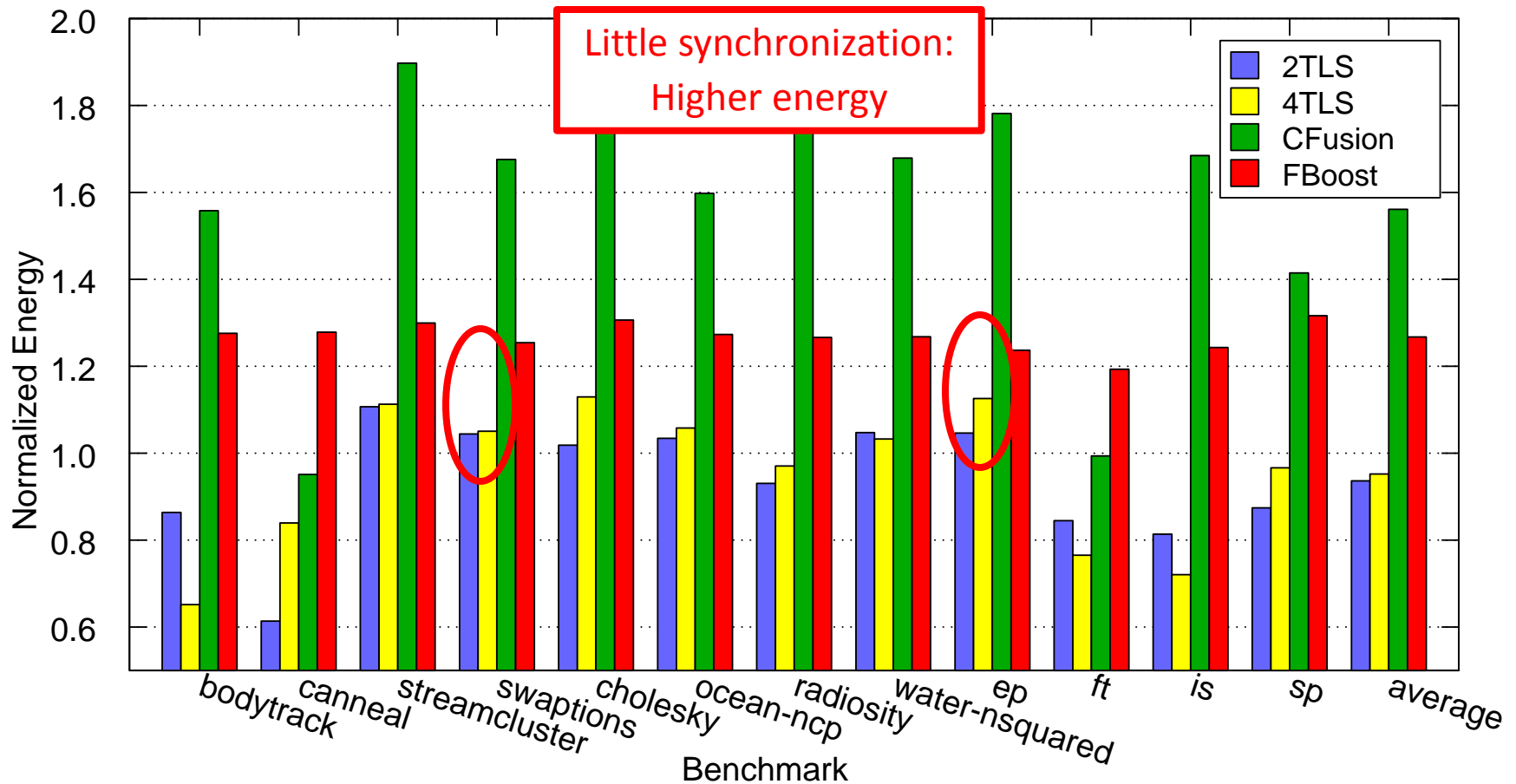
Energy

- Showing best performing point for each scheme



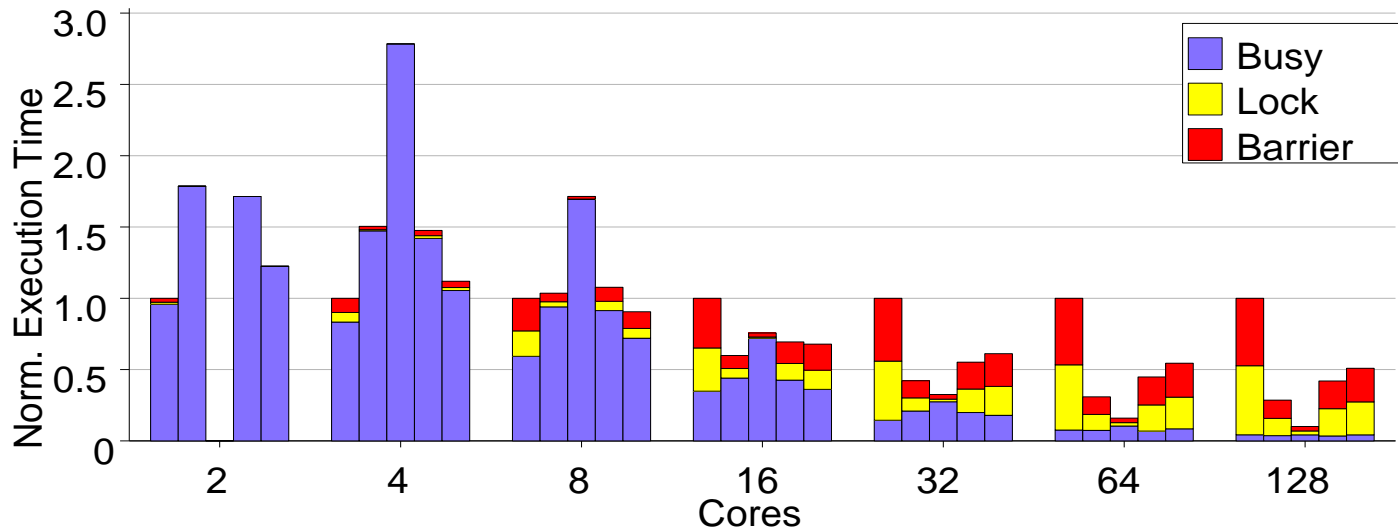
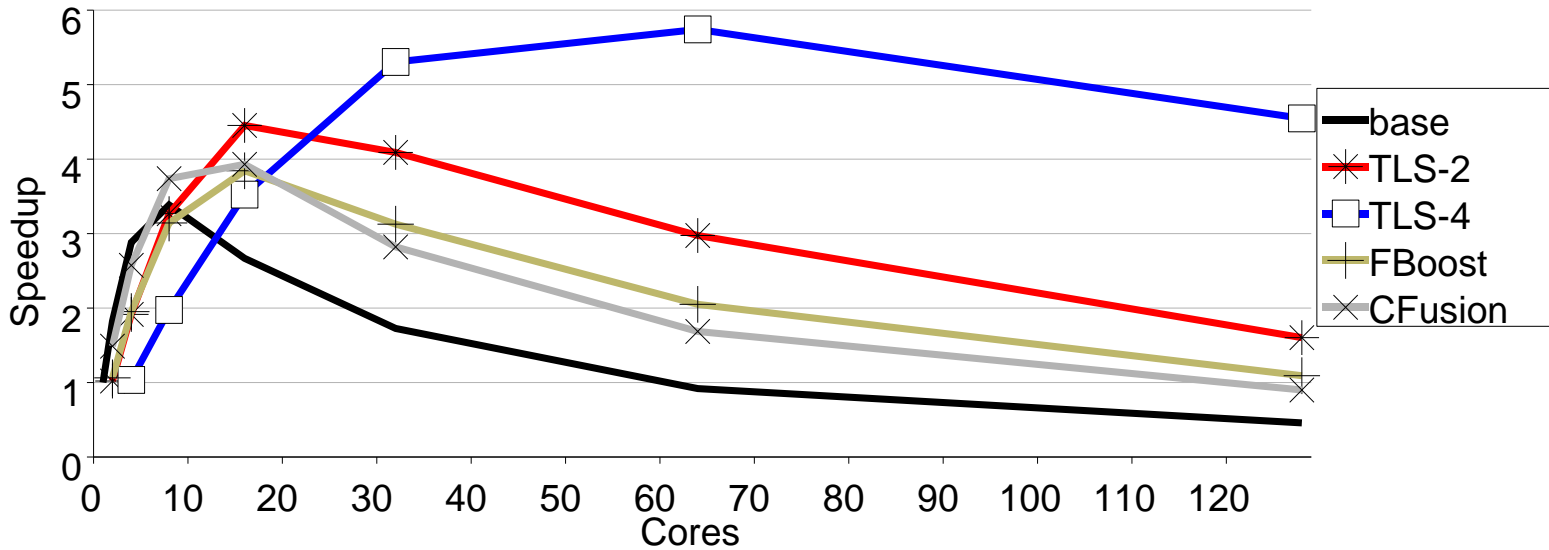
Energy

- Showing best performing point for each scheme



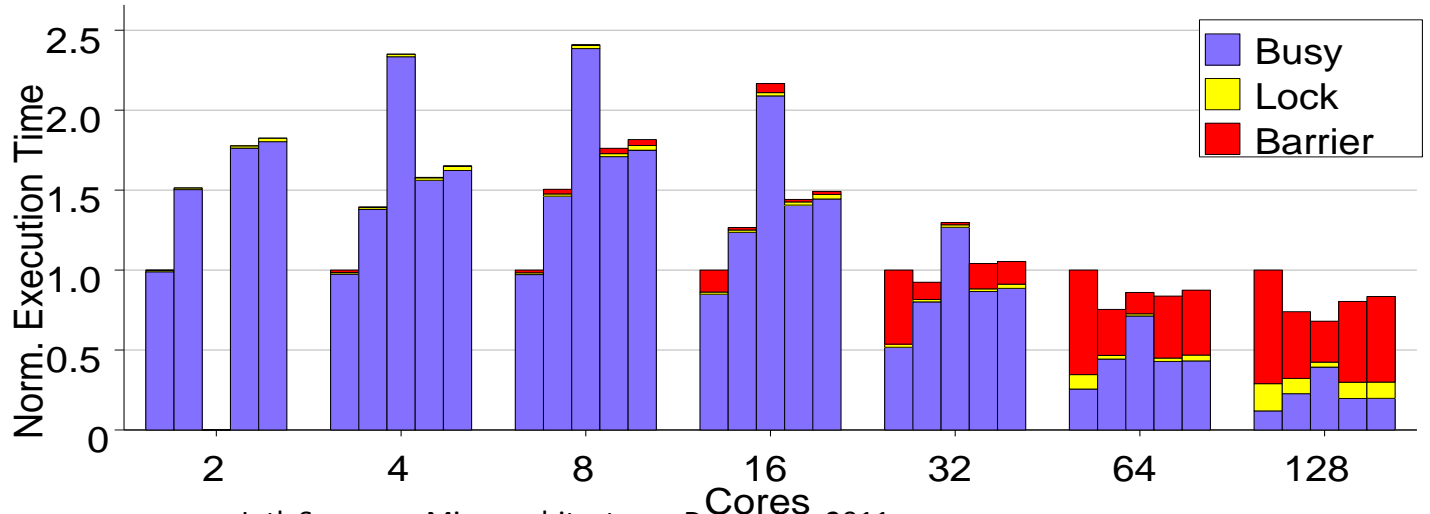
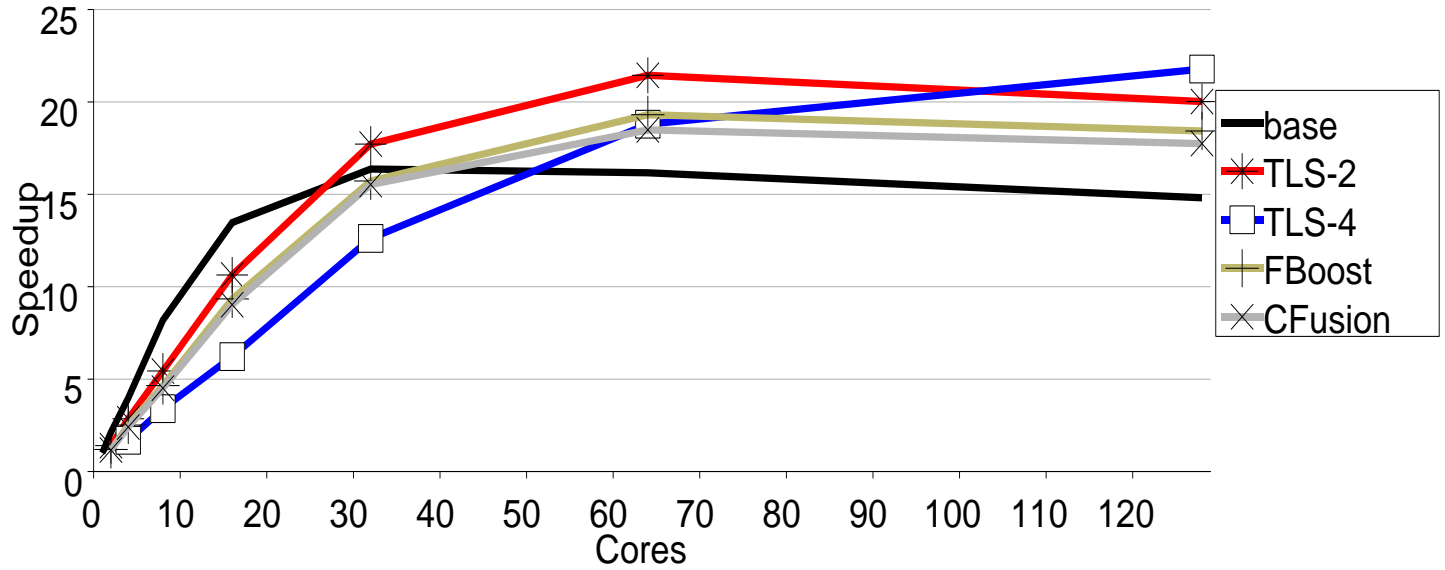
Serial/Critical Sections

is
NASPB



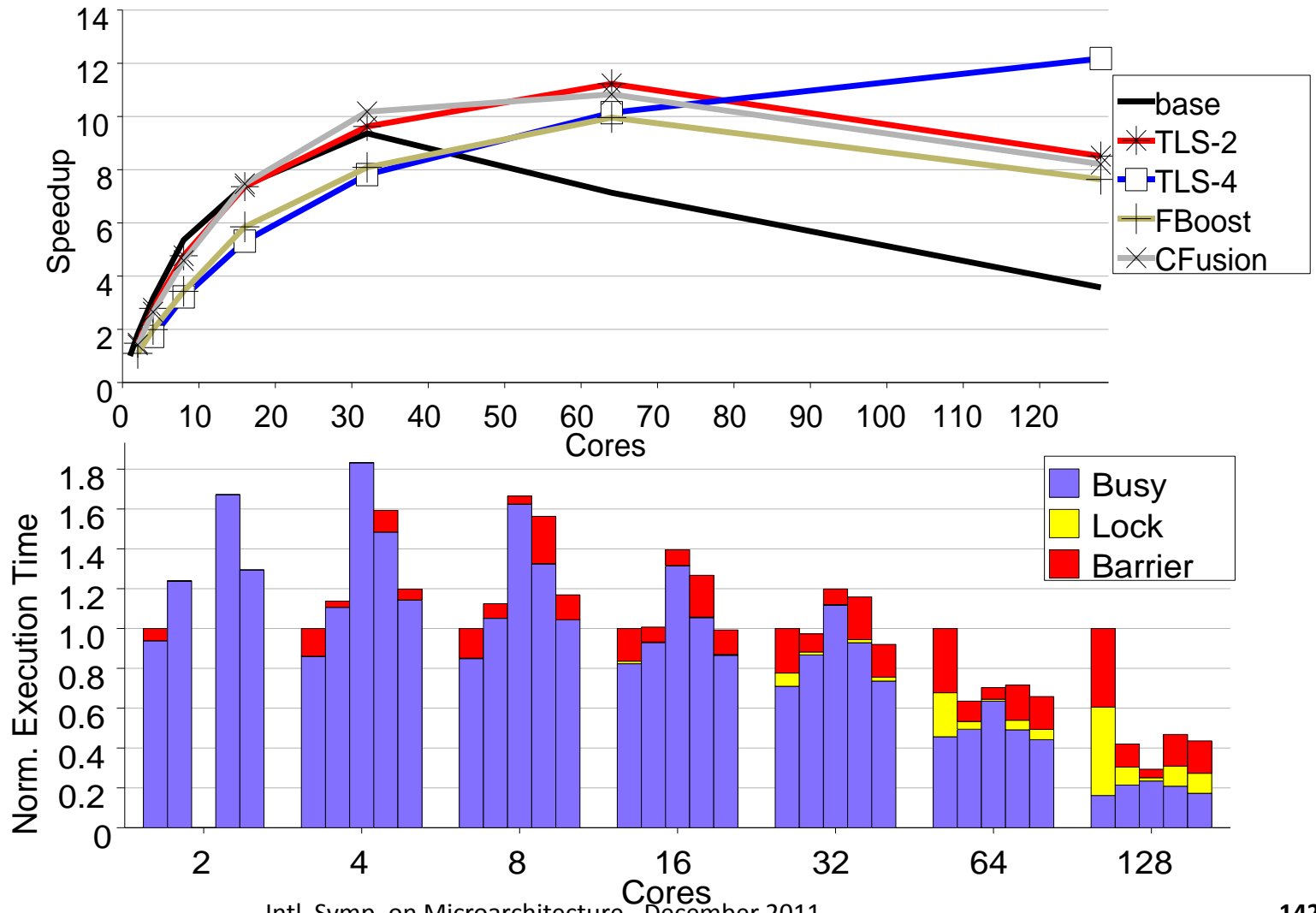
Load Imbalance

■ radiosity
SPLASH2



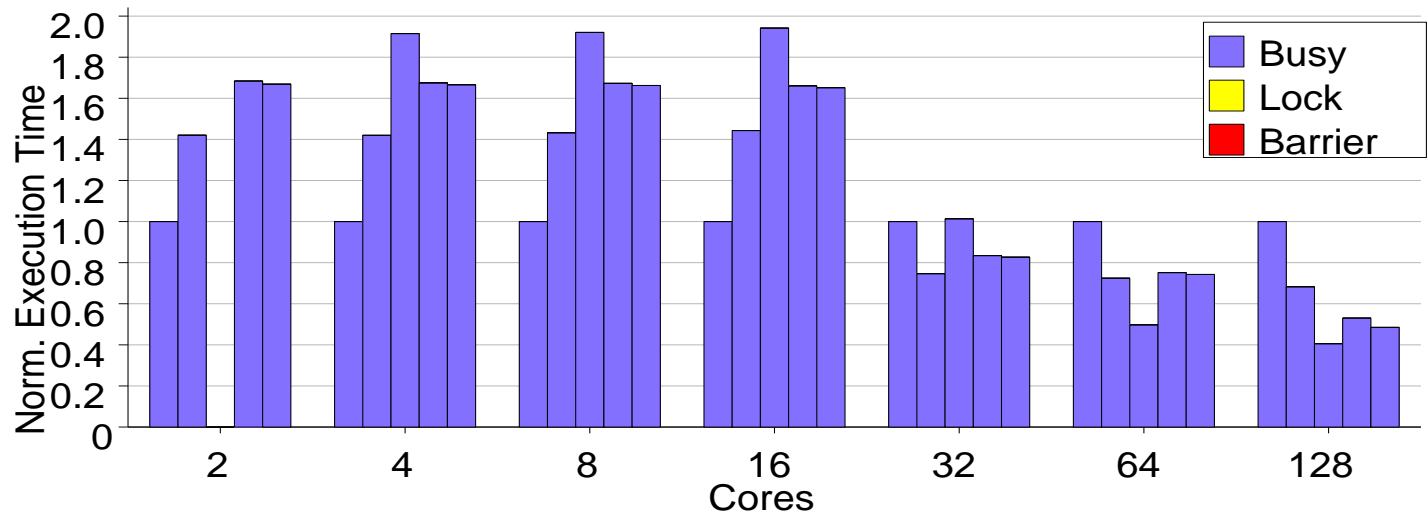
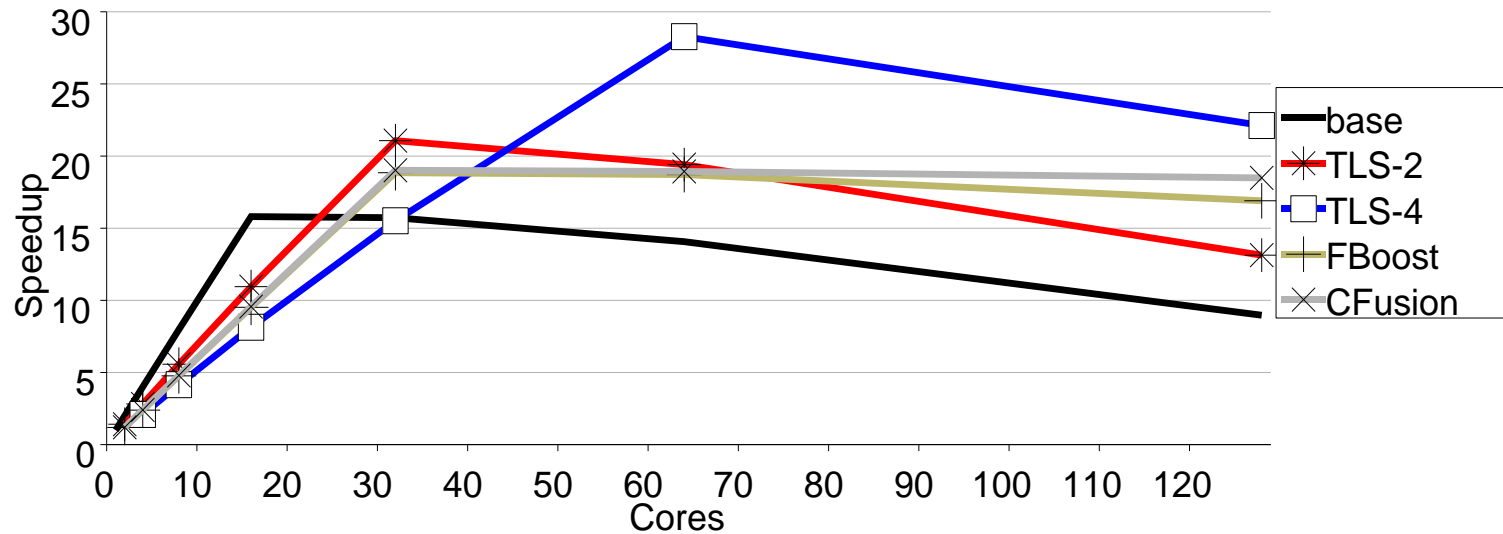
Synchronization Heavy

■ ocean
SPLASH2



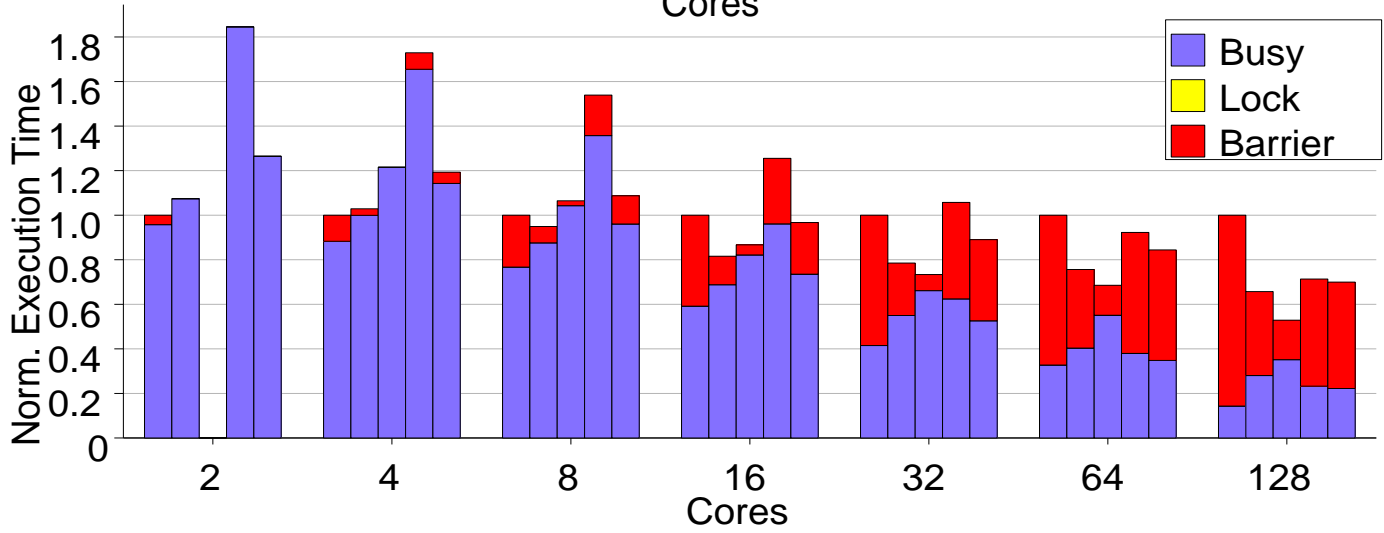
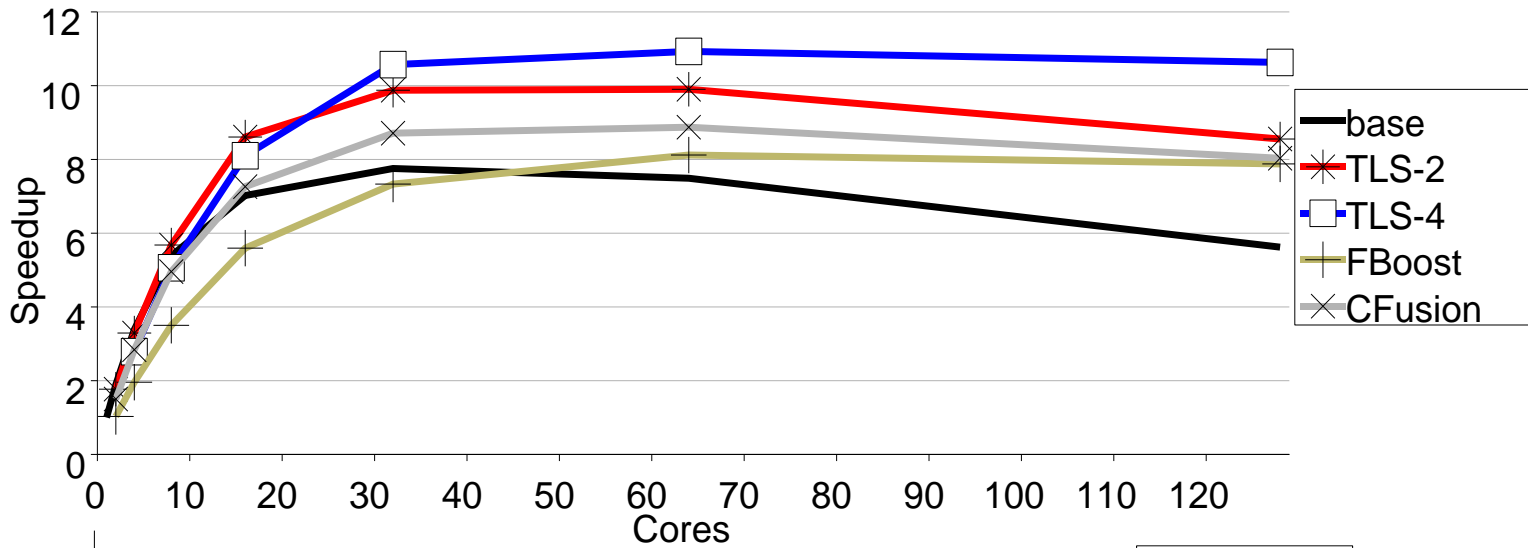
Coarse-Grain Partitioning

■ swaptions
PARSEC



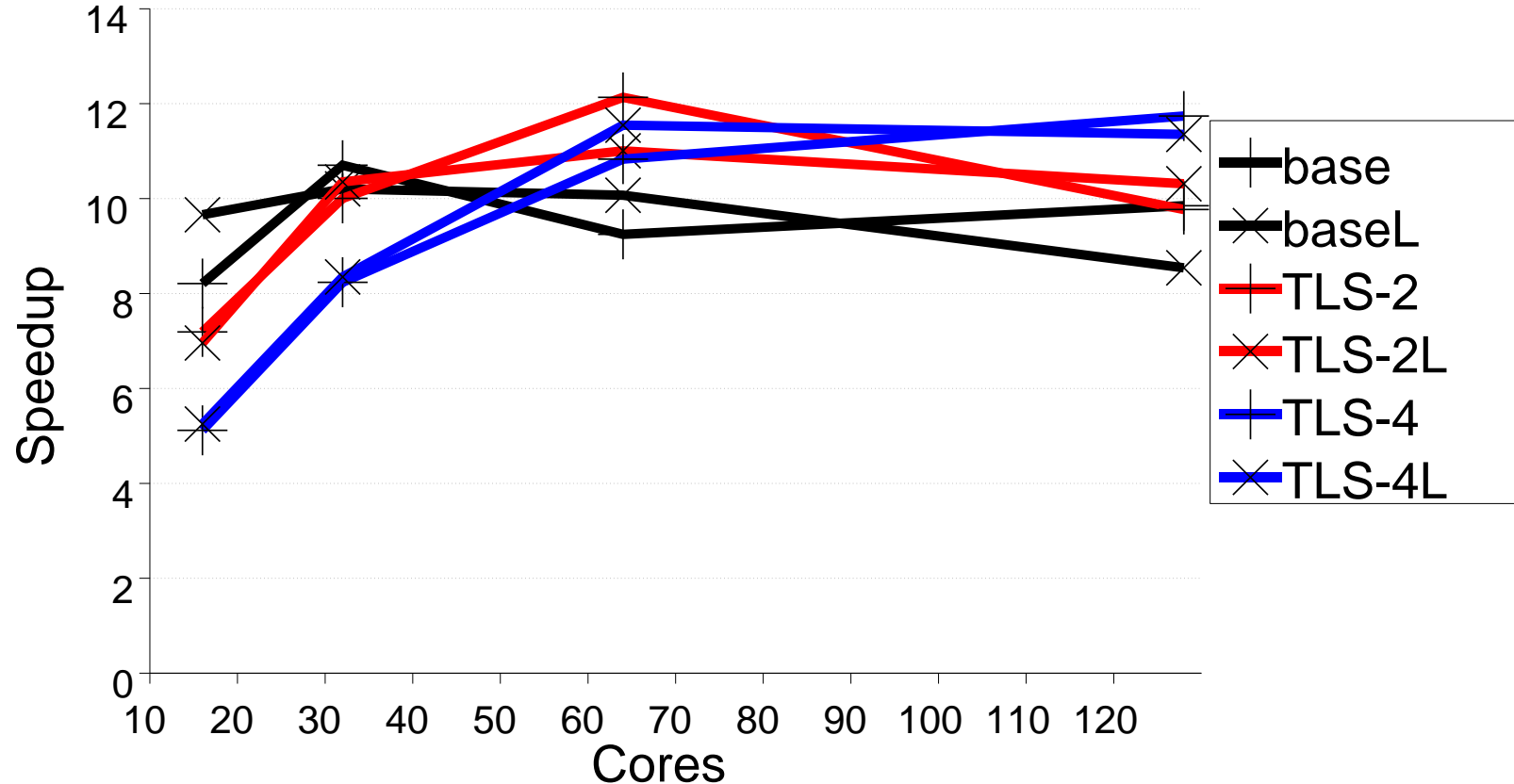
Poor Static Partitioning

■ sp
NASPB



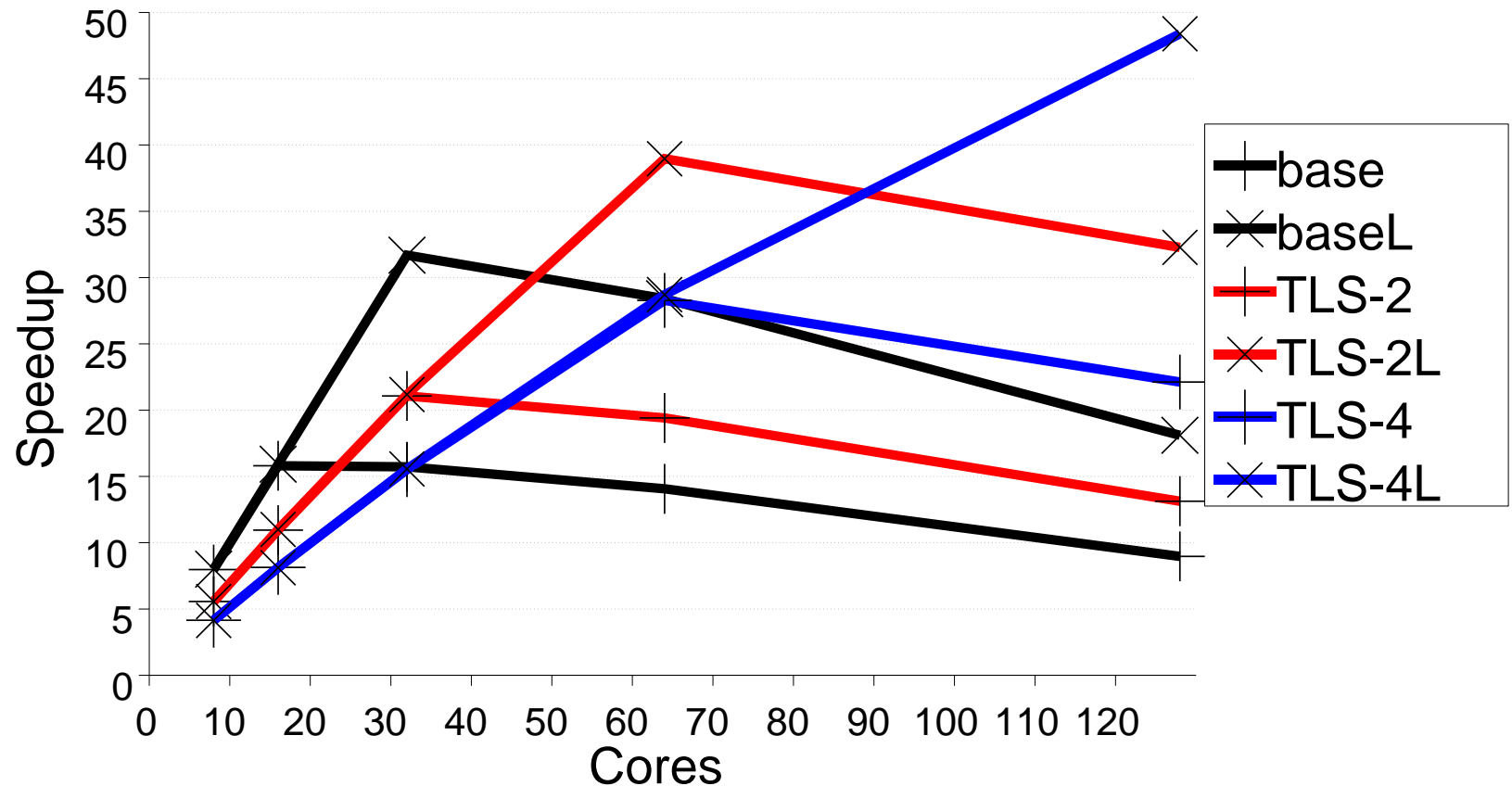
Effect of Dataset size

- Unchanged behavior: **cholesky**
- Also: canneal, ocean, ft, is, sp



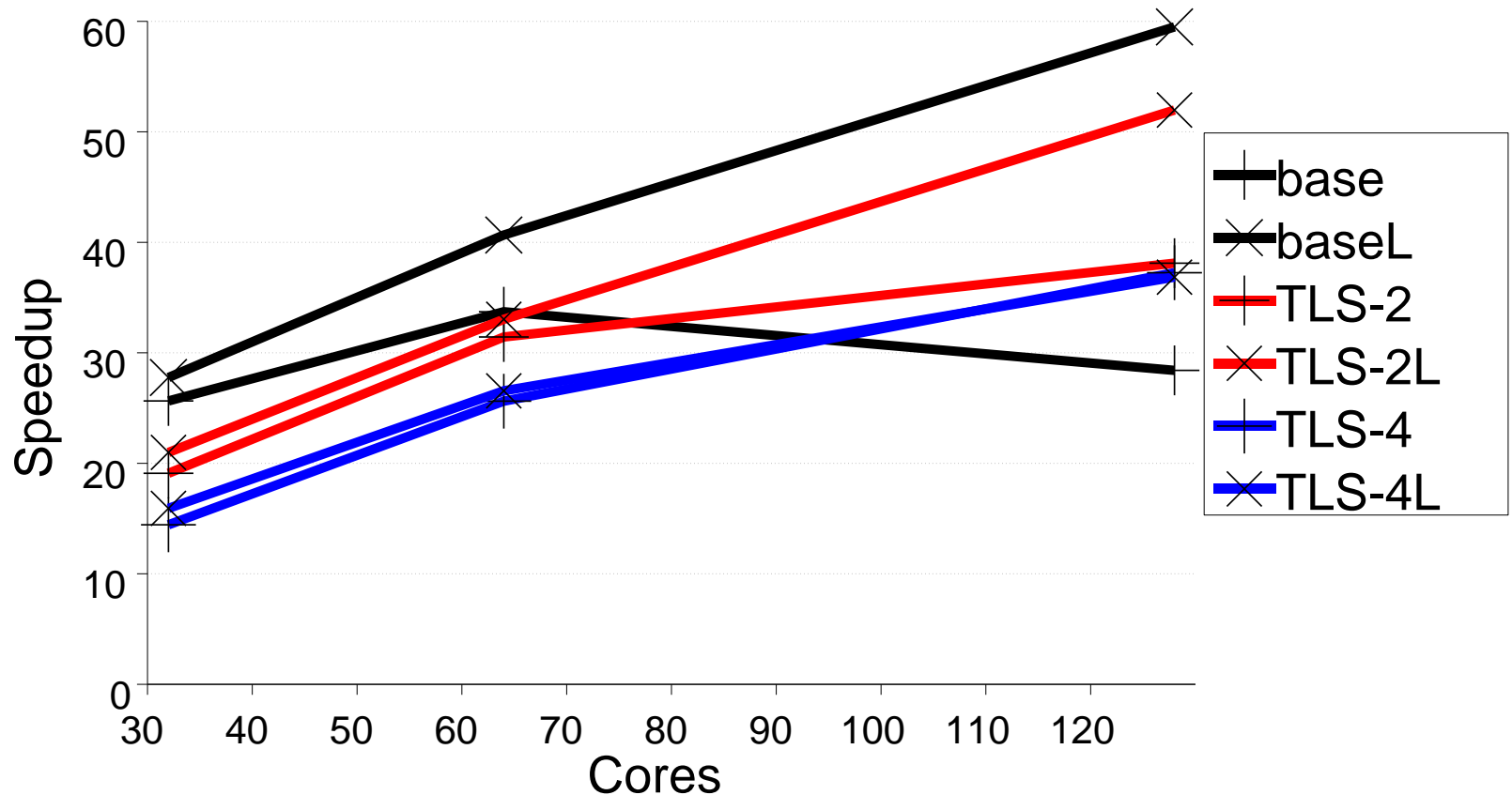
Effect of Dataset size

- Improved scalability, but TLS boost remains: **swaptions**
- Also: bodytrack, radiosity, ep



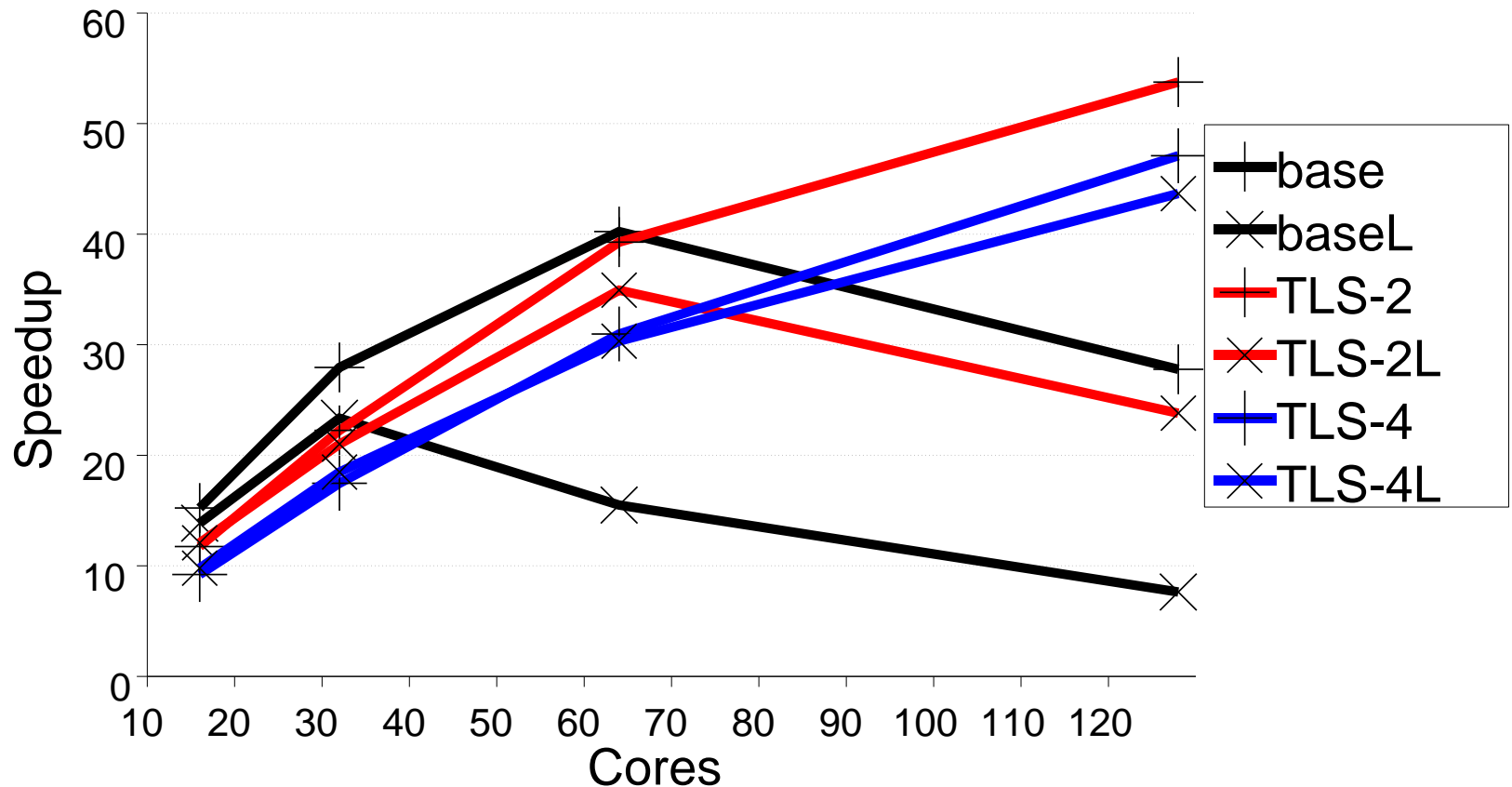
Effect of Dataset size

- Improved scalability, lessened TLS boost: **streamcluster**



Effect of Dataset size

- Worse scalability, even better TLS boost: **water**



Outline

- Introduction
- Motivation
- Proposal
- Evaluation Methodology
- Results
- Conclusions

Conclusions

- Multicores and many-cores are here to stay
 - Parallel programming essential to exploit new hardware
 - Some coarse-grain parallel programs do not scale
 - Enough nested parallelism to improve scalability
- Proposed speculative parallelization through implicit speculative threads on top of explicit threads:
 - Significant scalability improvement of 40% on avg
 - No increase in total energy consumptions
 - Presented an auto-tuning mechanism to dynamically choose the number of threads that performs within 6% of the oracle

Complementing User-Level Coarse-Grain Parallelism with Implicit Speculative Parallelism

Nikolas Ioannou, Marcelo Cintra

School of Informatics
University of Edinburgh



Related Work

- [von Praun PPOPP'07] Implicit ordered transactions
- [Kim Micro'10] Speculative Parallel-stage Decoupled Software Pipelining
- [Ooi ICS'01] Multiplex
- [Madriles ISCA'09] Anaphase
- [Rajwar MICRO'01],[Martinez ASPLOS'02] Speculative Lock Elision
- [Moravan ASPLOS'06], etc., Nested transactional memory

Bibliography

- [Intl'08] Intel Corp. Intel Turbo Boost Technology in Intel Core Microarchitecture (Nehalem) Based Processors, White Paper, 2008
- [Ipek ISCA'07] Ipek et al. Core fusion: Accommodating software diversity in chip multiprocessors
- [von Praun PPOPP'07] C. von Praun et al. Implicit parallelism with ordered transactions, PPOPP 2007
- [Kim Micro'10] Scalable speculative parallelization in commodity clusters, MICRO, 2010
- [Ooi ICS'01] C.-L Ooi et al. Multiplex: Unifying conventional and speculative thread-level parallelism on a chip multiprocessor, ICS 2001
- [Madriles ISCA'09] C. Madriles et al. Boosting single-thread performance in multi-core system through fine-grain multi-threading. ISCA 2009

Bibliography

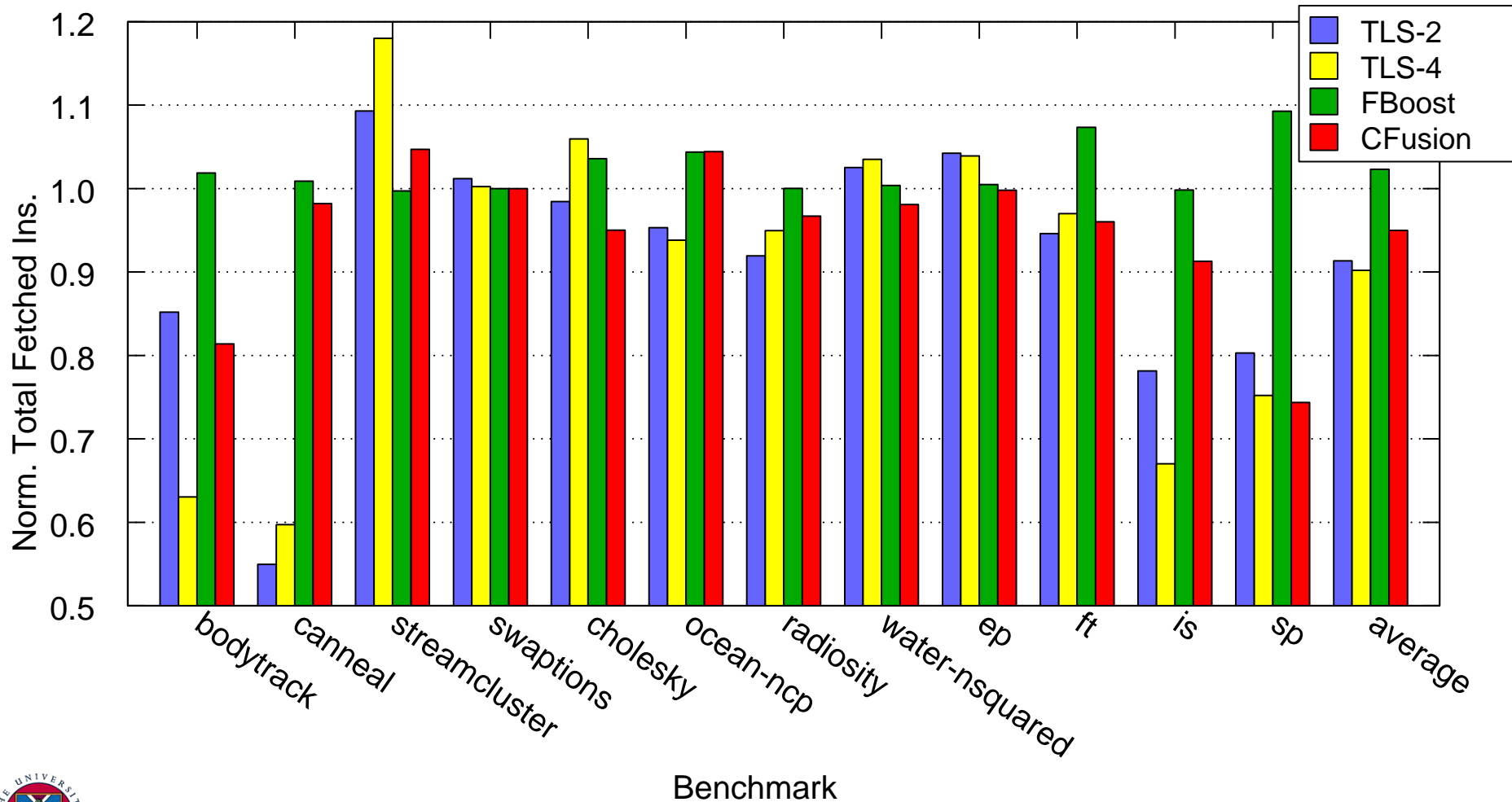
- [Rajwar MICRO'01] R. Rajwar and J.R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. MICRO 2001
- [Martinez ASPLOS'02] J. Martinez and J. Torellas. Speculative synchronization: Applying thread-level speculation to explicitly parallel applications. ASPLOS 2002
- [Moravan ASPLOS'06] Supporting nested transactional memory in logtm. ASPLOS 2006
- [Curtis-Maury PACT'08] Prediction models for multi-dimensional power-performance optimization on many-cores.

Benchmark details

Benchmark	Description	Input Sizes		Coverage of Speculative Regions ^a	Types of Speculation
		Normal	Large		
PARSEC					
bodytrack	Computer Vision	sequenceB_1	sequenceB_2	59%	LLS
canneal	Chip Design	100000.nets	200000.nets	99%	LLS
streamcluster	Data Mining	4K	8K	92%	LLS
swaptions	Financial Analysis	16	32	80%	LLS,MLS
SPLASH2					
cholesky	Sparse Matrix Multiplication	tk15	tk29	81%	LLS
ocean	Ocean Current Simulation	130	258	87%	LLS,MLS
radiosity	Graphics Rendering	test	room	69%	LLS,MLS
water	Molecular Dynamics	512	1000	99%	LLS
NAS OpenMP					
ep	Random Number Generator	1M	4M	100%	LLS,MLS
ft	3D FFT PDE	128K	512K	42%	LLS
is	Integer Sort	65K	1M	4%	LLS
sp	3D Fluid Dynamics	36	64	88%	LLS



Fetches Instructions



Failed Speculation

