Automatically Mapping Code on an Intelligent Memory Architecture

Jaejin Lee[†], Yan Solihin[‡], Josep Torrellas[‡]
[‡]University of Illinois at Urbana-Champaign
[†]Michigan State University
http://iacoma.cs.uiuc.edu/flexram

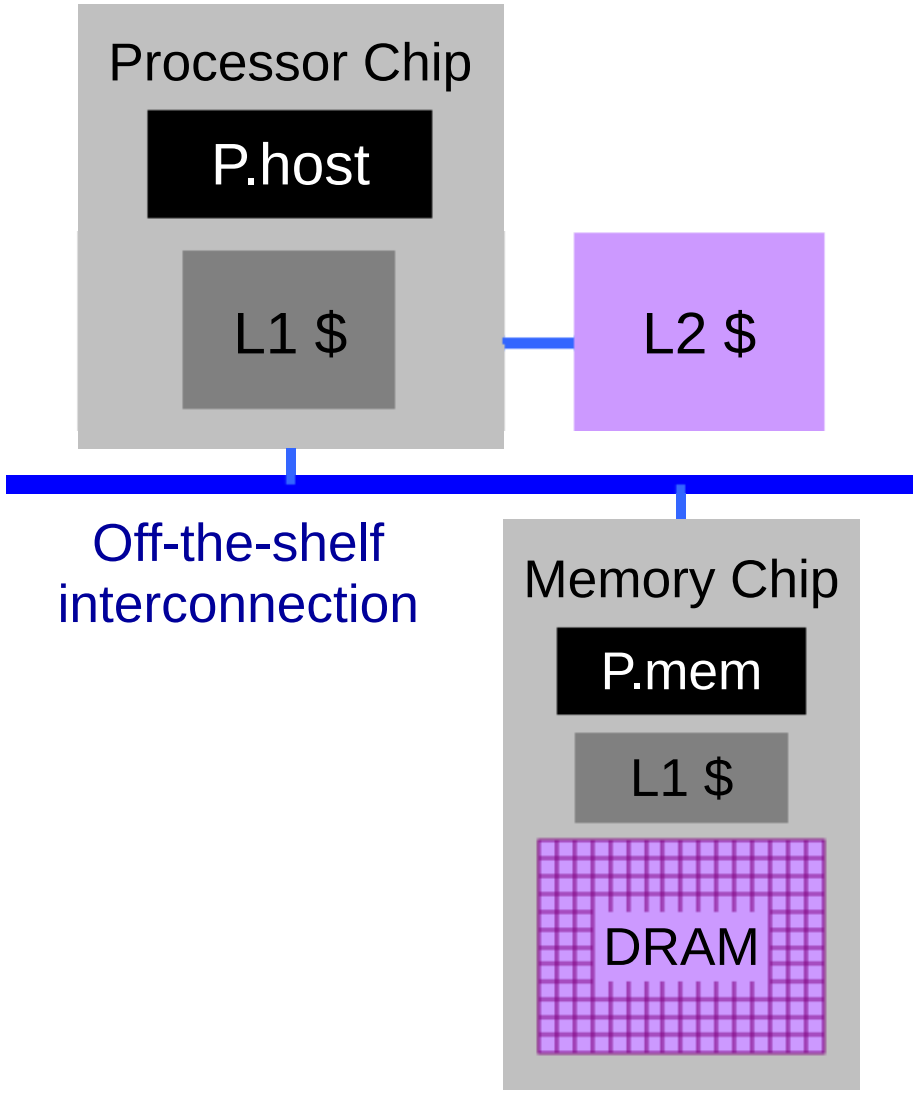
Overview

- Intelligent Memory Architecture
- Related Work
- Goals of This Study
- Mapping Algorithms
- Evaluation
- Conclusions

The Intelligent Memory Architecture

- Processors In Memory (PIM) Architecture Integrates processor logic and DRAM on a single chip.
- Two approaches:
  - IRAM, Shamrock, RAW, Smart Memories: the PIM architecture is main processing unit in the system
  - Active Pages, DIVA, FlexRAM: the PIM chips replace memory chips in the system (Intelligent Memory Architecture)
- The Intelligent Memory Architecture has a heterogeneous mix of processors.

The Intelligent Memory Architecture (cont'd)



- P.host: a wide-issue superscalar with a deep cache hierarchy.
- P.mem: a simple, narrow-issue superscalar with only a small cache.
- Compiler controlled cache coherence
  - P.host writeback dirty lines that might be read by P.mem.
  - P.host invalidates lines that might be written by P.mem.

Related Work

- Many different type of PIM architectures: IRAM, Shamrock, RAW, Smart Memories, Active Pages, DIVA, FlexRAM, etc.
- Previous work for Intelligent Memory architectures largely depend on programmers and focuses on running sections of code on a set of identical memory processors.
- Previous work in exploiting parallelism in a heterogeneous environment (Globus, Legion) has bigger granularity than ours and targets highly-distributed systems.

The goal of this study

- How to automatically program the Intelligent Memory Architecture?
  - A combination of static and run-time algorithms.
  - Partitioning code into smaller sections (*basic partitioning and advanced partitioning*).
  - Mapping the sections onto the best processor (*processor affinity estimation*).
  - Overlapping executions of the sections if possible.
- The algorithms are adaptive to the overheads.

Basic Partitioning

- Finds code sections (basic modules) that are easy to extract and have
  - homogeneous computing and memory behaviors
  - good locality.
- A basic module is a loop nest, where
  - each nesting level has only one loop
  - may span several subroutine levels.

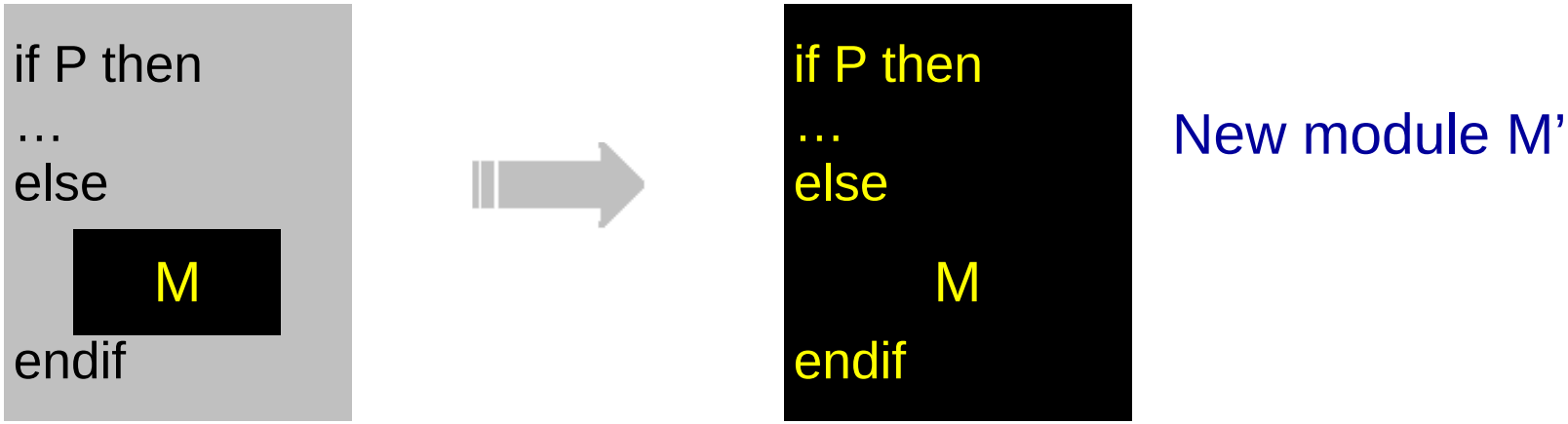
Basic Partitioning (cont'd)

```
N1 = N*2
DO I=1, N1
N2 = X * 4
DO J = 1, N2
X = …
A(J,I) = …
ENDDO
IF (X .LT. 1.0) THEN
X = …
ENDIF
ENDDO
C(N) = …
DO K = 1, N-1
B(K) = C(K+1)
ENDDO
```

```
N1 = N*2
DO I=1, N1
N2 = X * 4
DO J = 1, N2
X = …
A(J,I) = …
ENDDO
IF (X .LT. 1.0) THEN
X = …
ENDIF
ENDDO
C(N) = …
DO K = 1, N-1
B(K) = C(K+1)
ENDDO
```

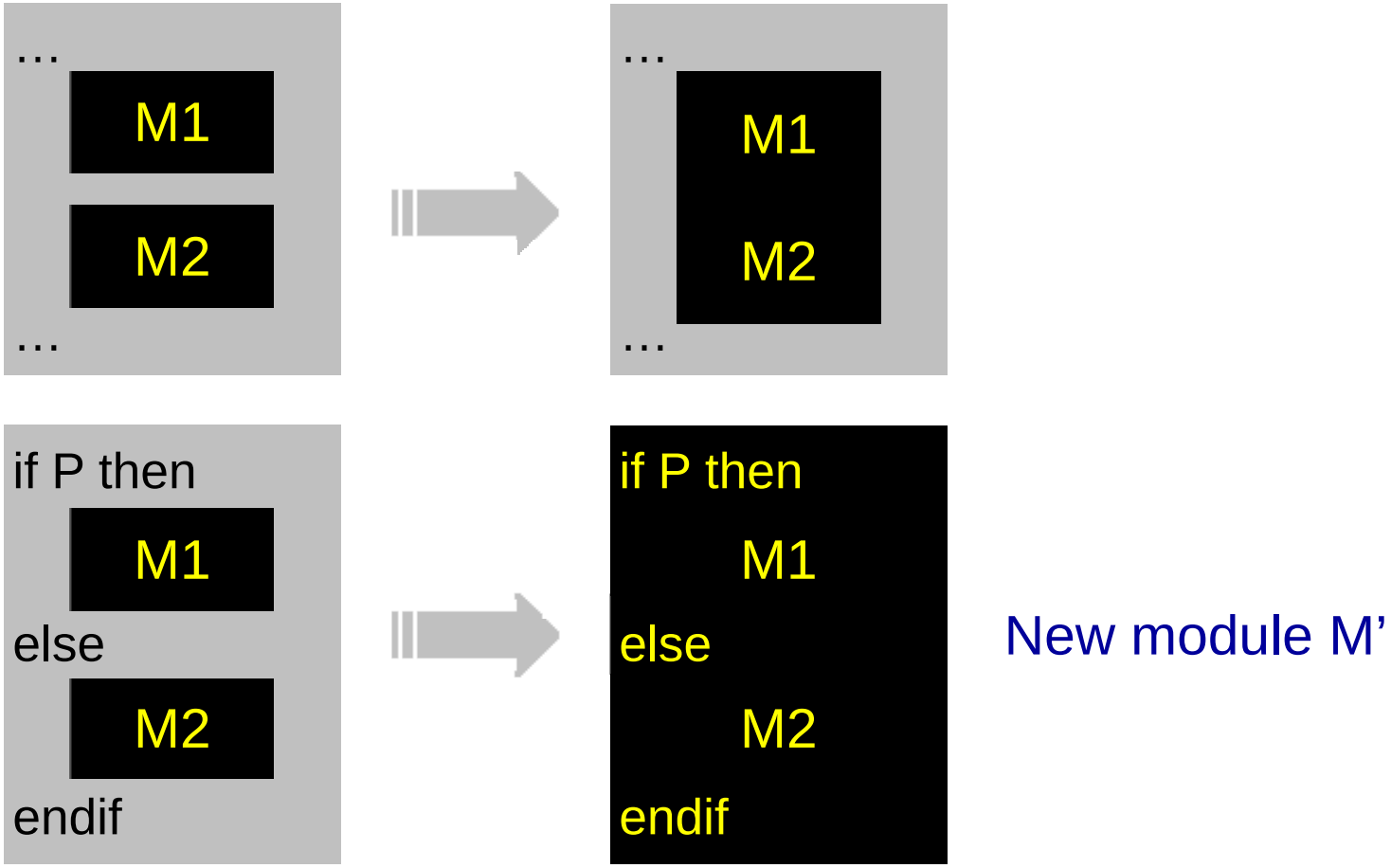
Advanced Partitioning

- Increases the grain size of the module, possibly reducing uniformity resulting in compound modules.
- Repeatedly applying *expansion* and *combining* steps.
- Expansion: similar to the basic partitioning



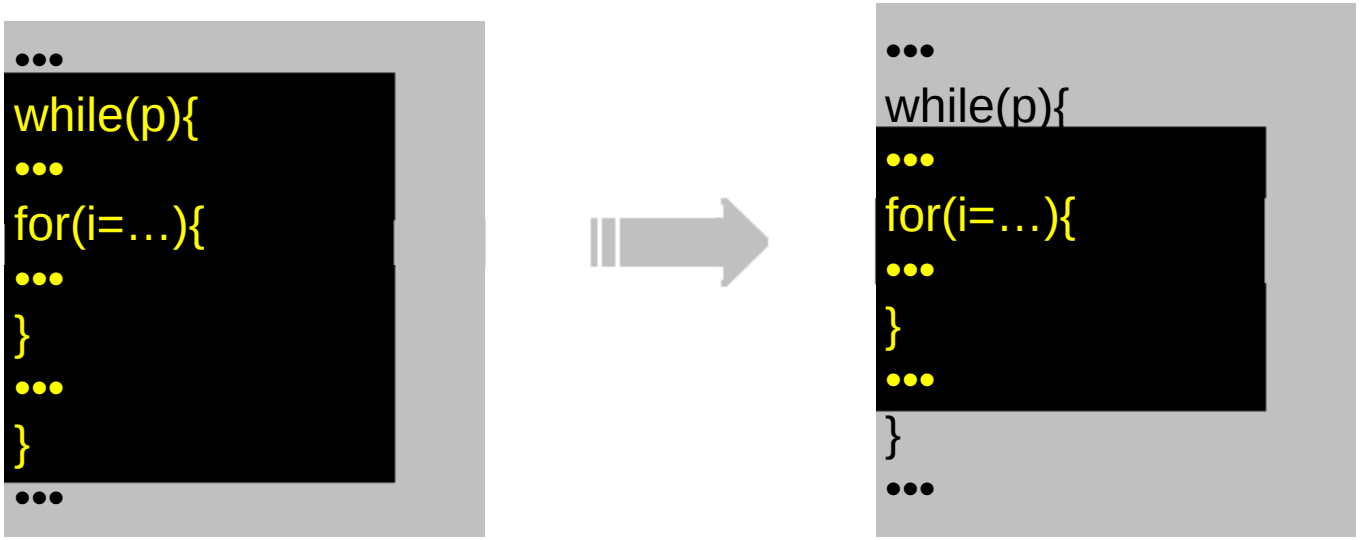
Advanced Partitioning (cont'd)

- Combining: two adjacent modules with the same affinity are combined into a new module.



Advanced Partitioning with Retraction

- Compound modules resulting from advanced partitioning may be very large and invoked very few times (harder run-time adaptation).
- The algorithm selects advanced modules that are expected to be invoked only 1-2 times.
- Peel-off statements until it reaches an all enclosing loop or a set of disjoint loops.



Mapping (Static)

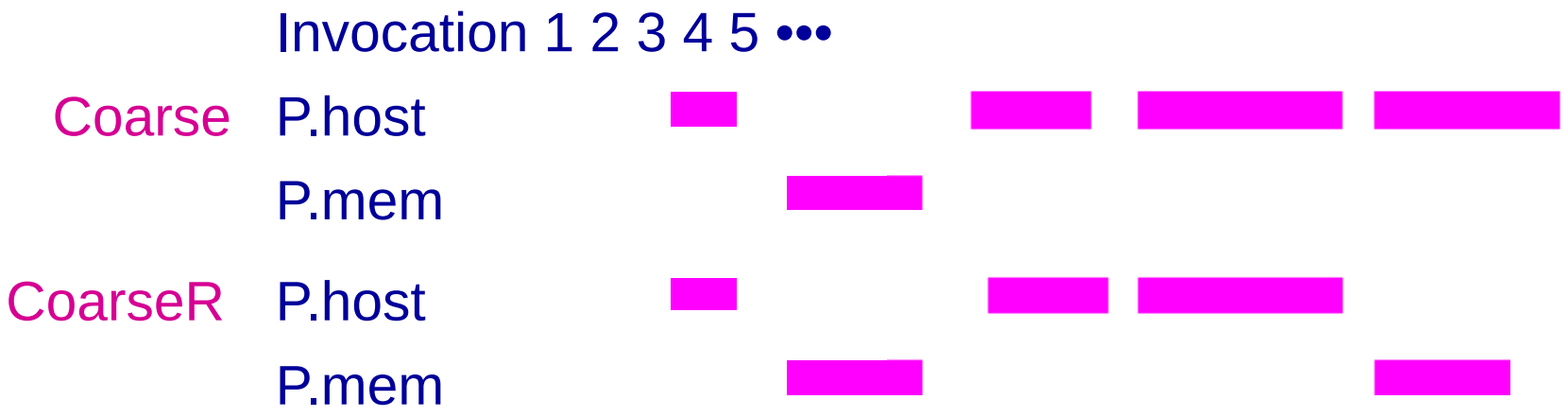
- Performance model (Delphi tool)
  - For numerical applications.
  - Execution time = $T_{comp} + T_{memstall}$

    $T_{comp} = max\ (T_{int} / N_{int}\ ,\ T_{fp} / N_{fp},\ T_{ldst} / N_{ldst}) + T_{other}$

    $T_{memstall} = \sum_{i=caches} (Miss\_penalty_i)$
  - Stack distance model for the number of misses.
- Profiling
  - For non-numerical applications
  - Gather execution time and the number of invocations for all modules and subroutines.

Mapping (Dynamic)

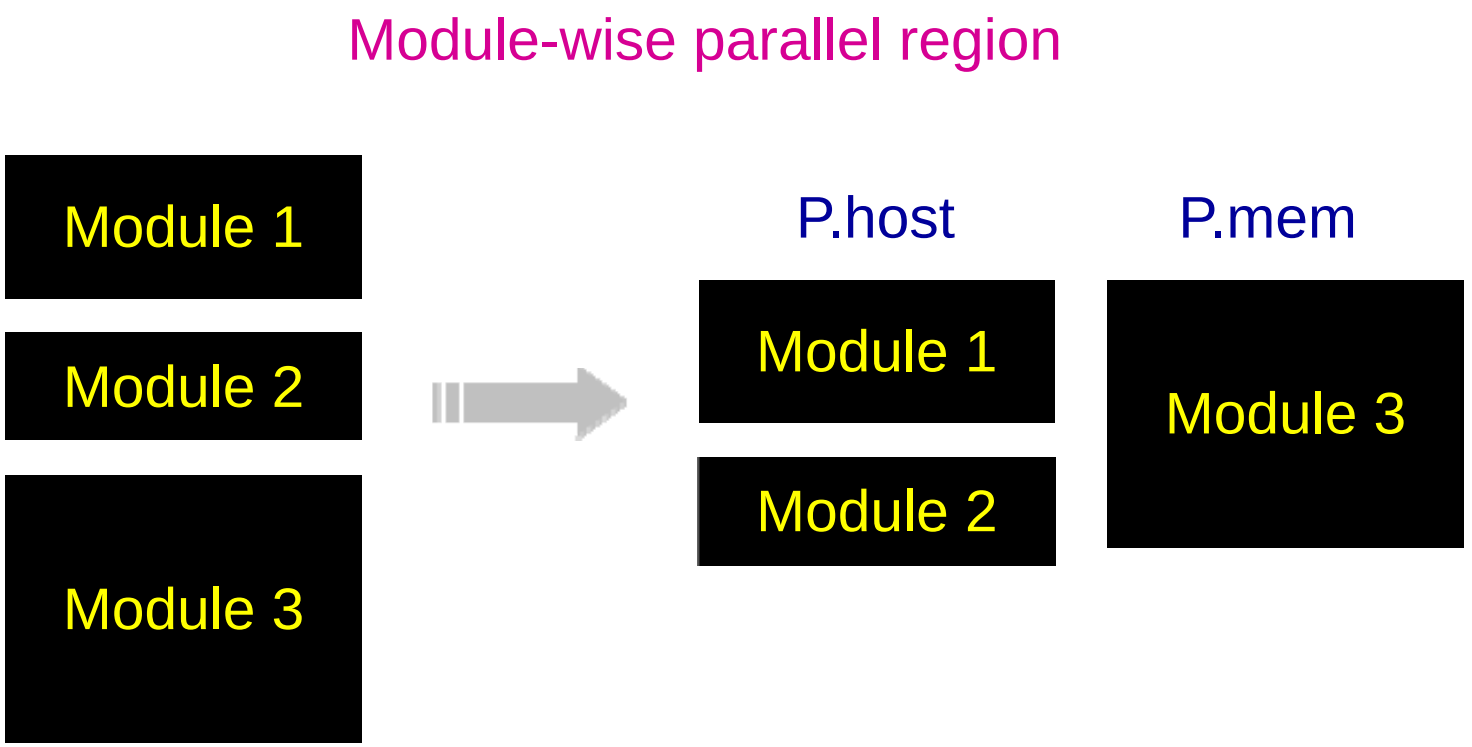
- Decision runs to determine affinity
- Coarse and CoarseR (adaptive to workload)



- For both basic modules and compound modules
- Fine and FineF are similar to Coarse and CoarseR, but the decision runs are the first two iterations of each invocation (high overhead).
  - Only for basic modules

Overlapping Execution

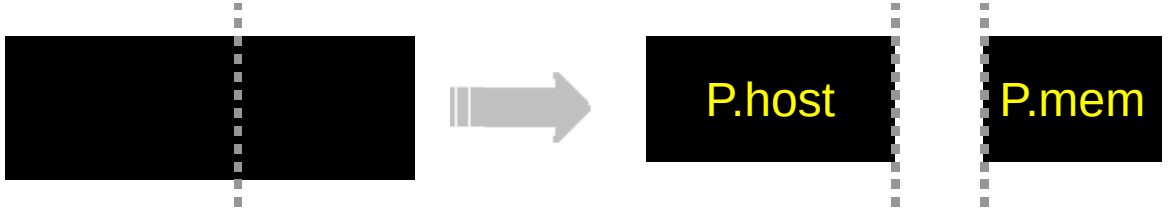
- With basic partitioning.
- Module-wise parallel region and module-wise serial region.



Module-wise Serial Region

- Static vs. Dynamic partitioning for overlapping execution to balance the load considering cache write-back and invalidation overheads.

Fully parallel



Distributable

Dopipe

```
do i=1,100
A(i) = A(i-1)+B(i)
C(i) = A(i)
enddo
```

```
P.host
do i=1,100
A(i) = A(i-1)+B(i)
if mod(i,4)=0 then
Writeback
Signal
endif
enddo
```

```
P.mem
do i=1,100
if mod(i+3,4)=0 then
Wait
endif
C(i) = A(i)
Enddo
```

Evaluation Environment

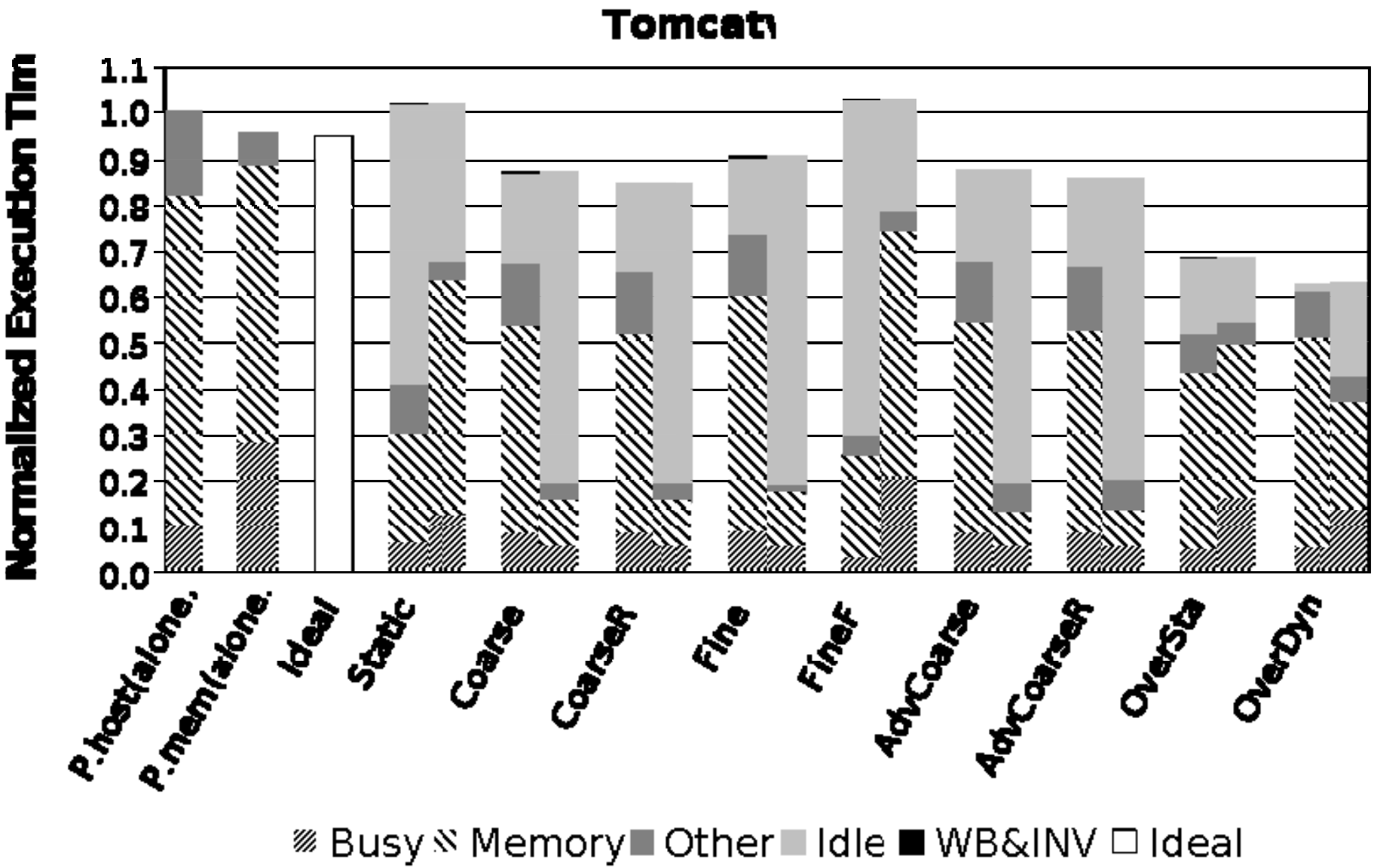
- Evaluated both numerical (by Polaris compiler) and non-numerical (by hand) applications.
- Mint based simulator.

Module	Parameter	Values
P.host::P.mem	Frequency Issue width Functional Units	800MHz :: 800MHz Out-of-order 6 :: In-order 2 4Int+4Fp+2Ld/St :: 2Int+2Fp+1Ld/St
P.host Caches	L1-Data L2-Data Write-back overhead Invalidation overhead	Write-through, 32KB, 2-cycle hit Write-back, 1MB (512KB for non-numerical apps.), 10-cycle hit 5 + 1 □ num_cache_lines (background) 5 + 1 □ num_cache_lines
P.mem Cache	L1-Data	Write-back, 16KB, 2-cycle hit
Memory and Bus	Memory Latency (cycles) Bus Type	160 from P.host, 21 from P.mem Split transaction, 16-B wide

Average Characteristics of Basic Modules

- Different applications have a different distribution of module affinity.

Averages	Numerical Applications	Non-numerical Applications
Total Modules	13.2 (99.1%)	41.8 (63.0%)
Parallel Modules Serial Modules	11.4 (70.9%) 1.8 (28.2%)	8.8 (1.3%) 33.0 (61.7%)
P.host Affinity P.mem Affinity	4.0 (37.0%) 9.2 (62.1%)	31.0 (38.9%) 37.8 (38.9%)
Average Number of Invocations	442.9	182,177
Average Module Size (P.host cycles)	4,570 K	477 K


Tomcat
Normalized Execution Tim
1.1
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
P.host(alone.
P.mem(alone.
Ideal
Static
Coarse
CoarseR
Fine
FineF
AdvCoarse
AdvCoarseR
OverSta
OverDyn
Busy
Memory
Other
Idle
WB&INV
Ideal

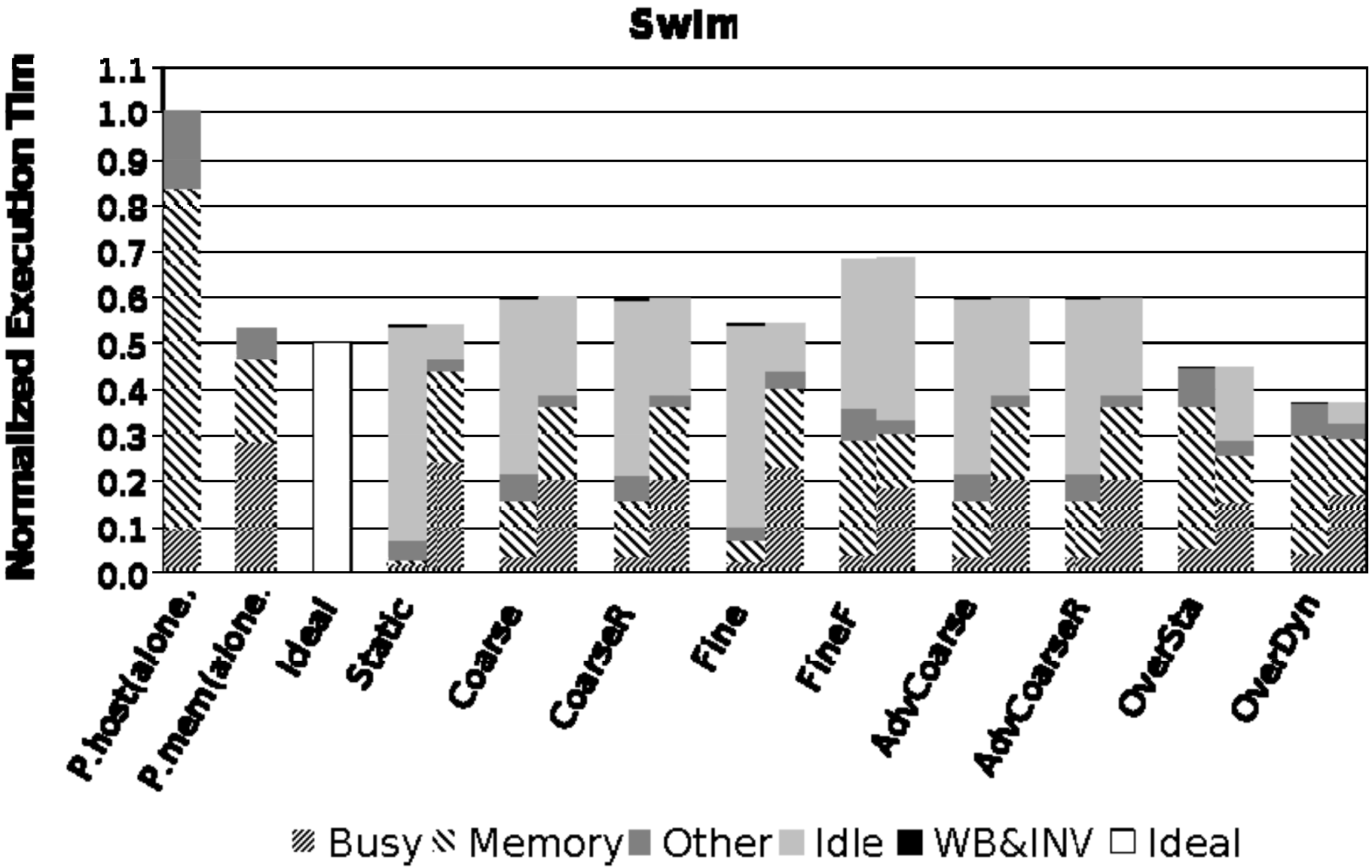
Overall Speedups

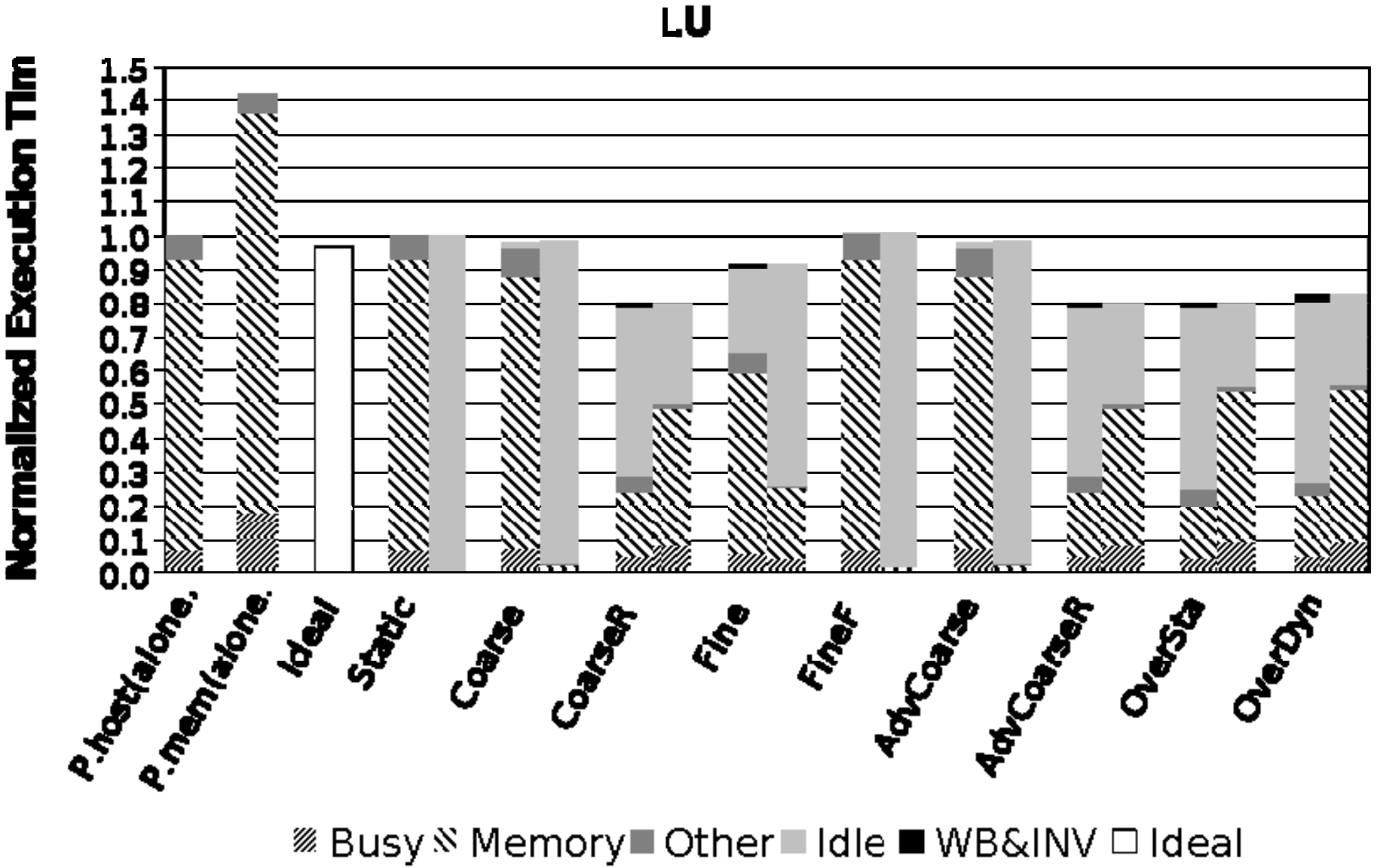
- Our algorithm delivers speedups that are comparable to the ideal speedup.

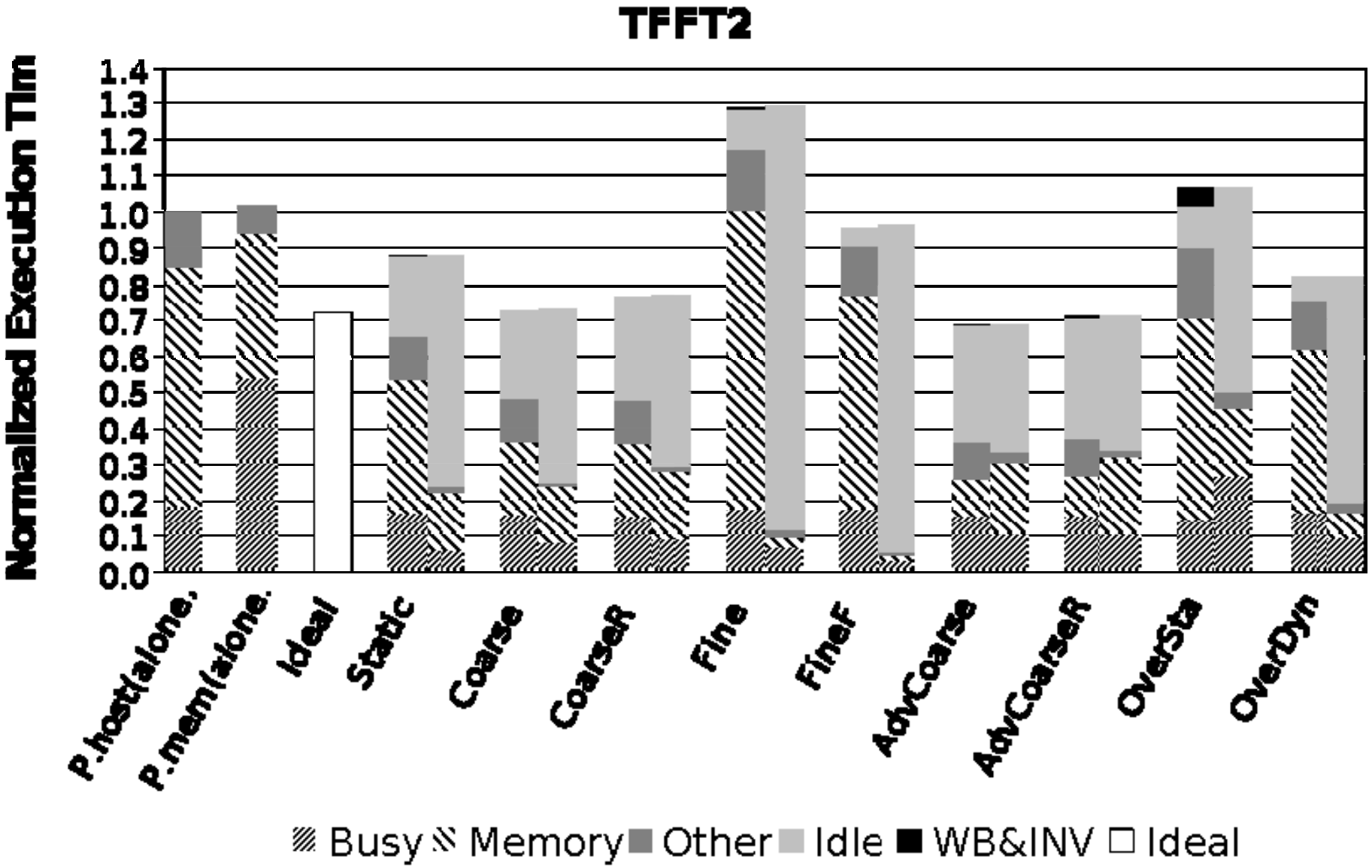
Apps.	P.host(alone) /AdvCoarseR	P.host(alone) /OverDyn	Amdahl's 2 P.hosts	2-processor SGI
Swim	1.67	2.71	2.00	1.85
Tomcatv	1.17	1.60	1.67	1.44
LU	1.26	1.22	1.04	0.99
TFFT2	1.42	1.22	1.91	0.80
Mgrid	1.05	1.55	1.94	1.47
Average	1.31	1.66	1.71	1.31
Bzip2	1.37	-	1.01	0.99
Mcf	1.37	-	1.01	1.00
Go	0.97	-	1.01	0.57
M88ksim	1.01	-	1.03	1.00
Average	1.18	-	1.02	0.89

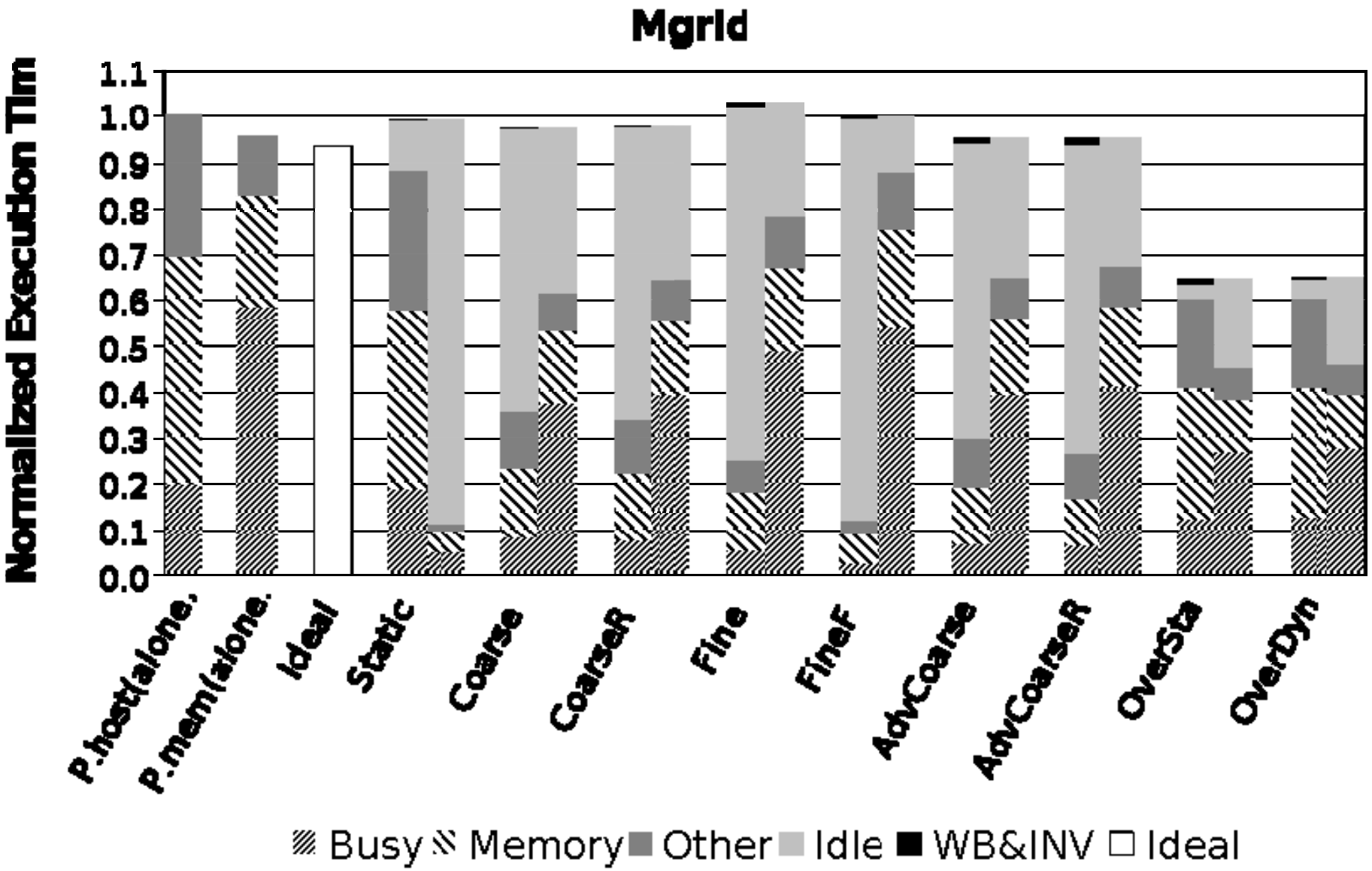
Conclusions

- Different applications have different computing and memory behaviors.
- By using a combination of static and dynamic algorithms, we achieve comparable speedups to the ideal speedup on the 2-host multiprocessor systems.
- A heterogeneous mix of processors can be exploited cost-effectively.


Swim
Normalized Execution Tim
1.1
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
P.host(alone.
P.mem(alone.
Ideal
Static
Coarse
CoarseR
Fine
FineF
AdvCoarse
AdvCoarseR
OverSta
OverDyn
Busy
Memory
Other
Idle
WB&INV
Ideal


LU
Normalized Execution Tim
1.5
1.4
1.3
1.2
1.1
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
P.host(alone.
P.mem(alone.
Ideal
Static
Coarse
CoarseR
Fine
FineF
AdvCoarse
AdvCoarseR
OverSta
OverDyn
Busy
Memory
Other
Idle
WB&INV
Ideal


TFFT2
Normalized Execution Tim
1.4
1.3
1.2
1.1
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
P.host(alone.
P.mem(alone.
Ideal
Static
Coarse
CoarseR
Fine
FineF
AdvCoarse
AdvCoarseR
OverSta
OverDyn
Busy
Memory
Other
Idle
WB&INV
Ideal


Mgrid
Normalized Execution Tim
1.1
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
P.host(alone.
P.mem(alone.
Ideal
Static
Coarse
CoarseR
Fine
FineF
AdvCoarse
AdvCoarseR
OverSta
OverDyn
Busy
Memory
Other
Idle
WB&INV
Ideal