Reducing Dynamic Compilation Overhead by Overlapping Compilation and Execution

P. Unnikrishnan
IBM Toronto, Canada
M. Kandemir and F. Li
Pennsylvania State University, US

Outline

- Introduction
- Overlapping compilation and execution
- Execution Model
- Experiments
- Conclusion

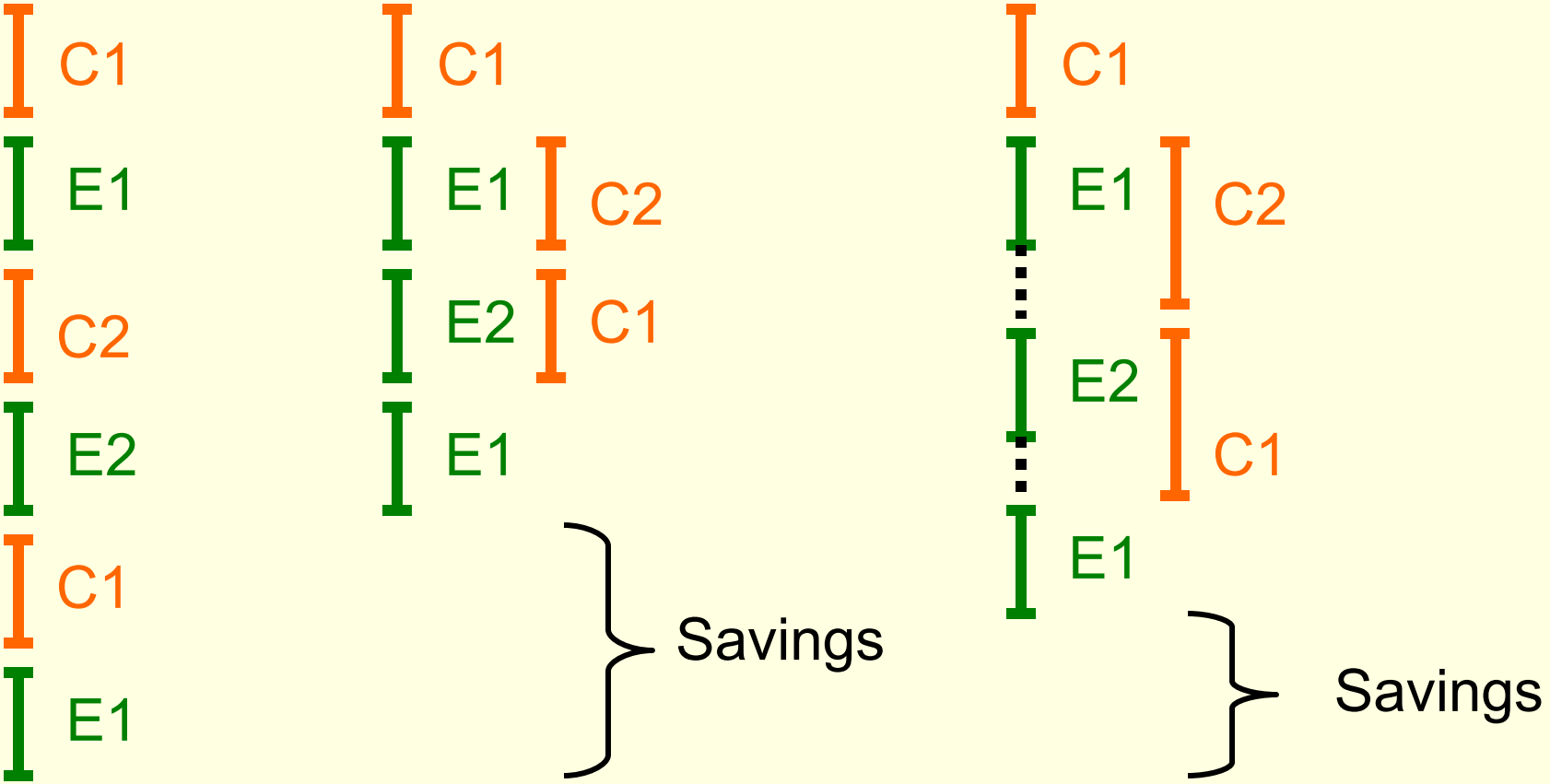
Introduction

- Dynamic compilation
  - Optimizing applications while they are executing
- Performance-oriented
  - Constant propagation through knowledge of variable values at runtime
  - Anticipating frequent future executions
- Energy-oriented
  - Application recompilation due to changing energy constraints
- Performance and energy overheads
- Our goal: reduce performance overhead of dynamic compilation

Introduction

- Overlapping dynamic compilation and application execution (compilation parallelization)
- Code fragment (module)
  - Loop nests, subroutines or several logically-related subroutines
- Strategy
  - Predict the next code fragment to be executed
  - Pre-compile that code fragment before it is actually needed

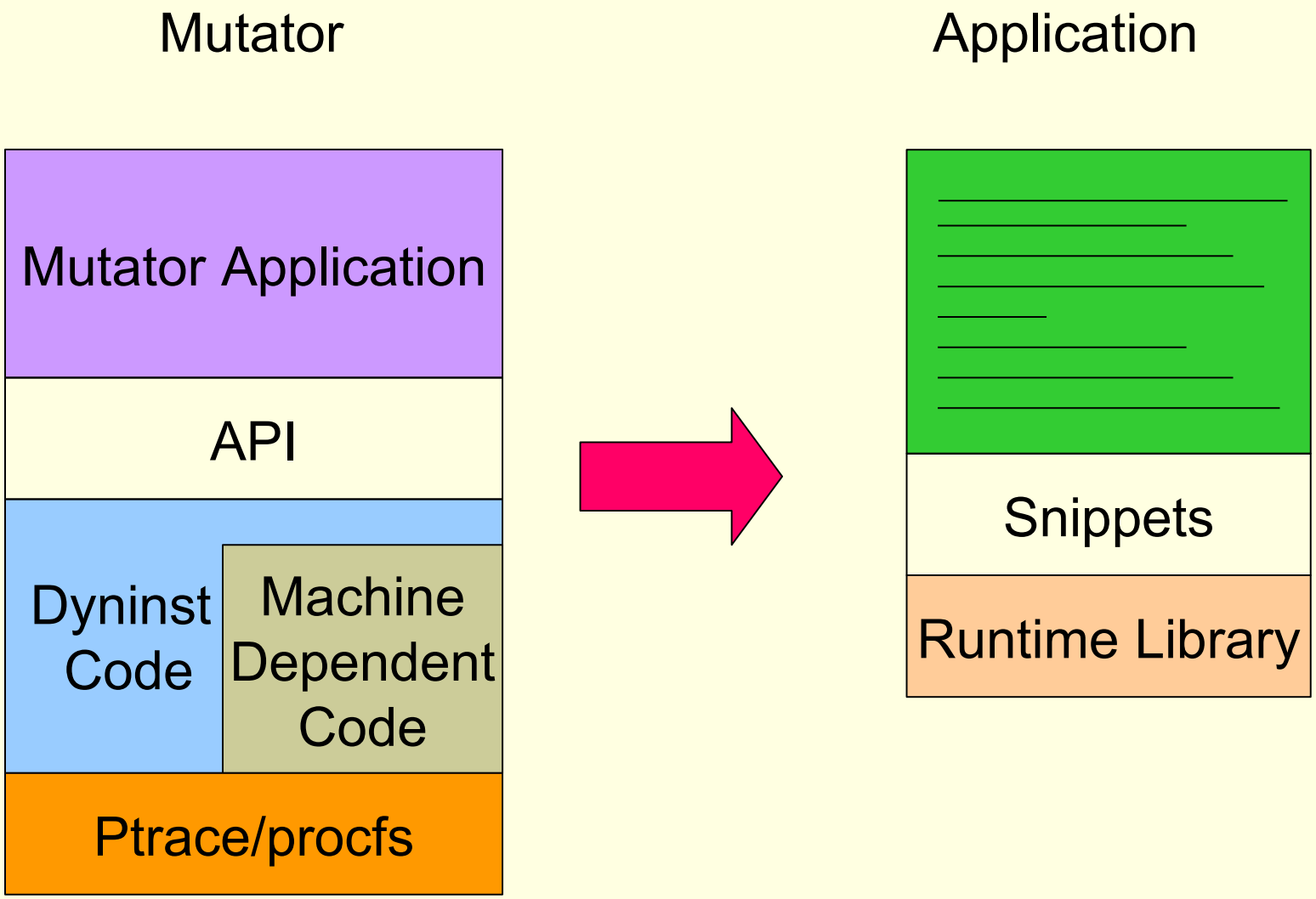
Overlapping compilation and execution



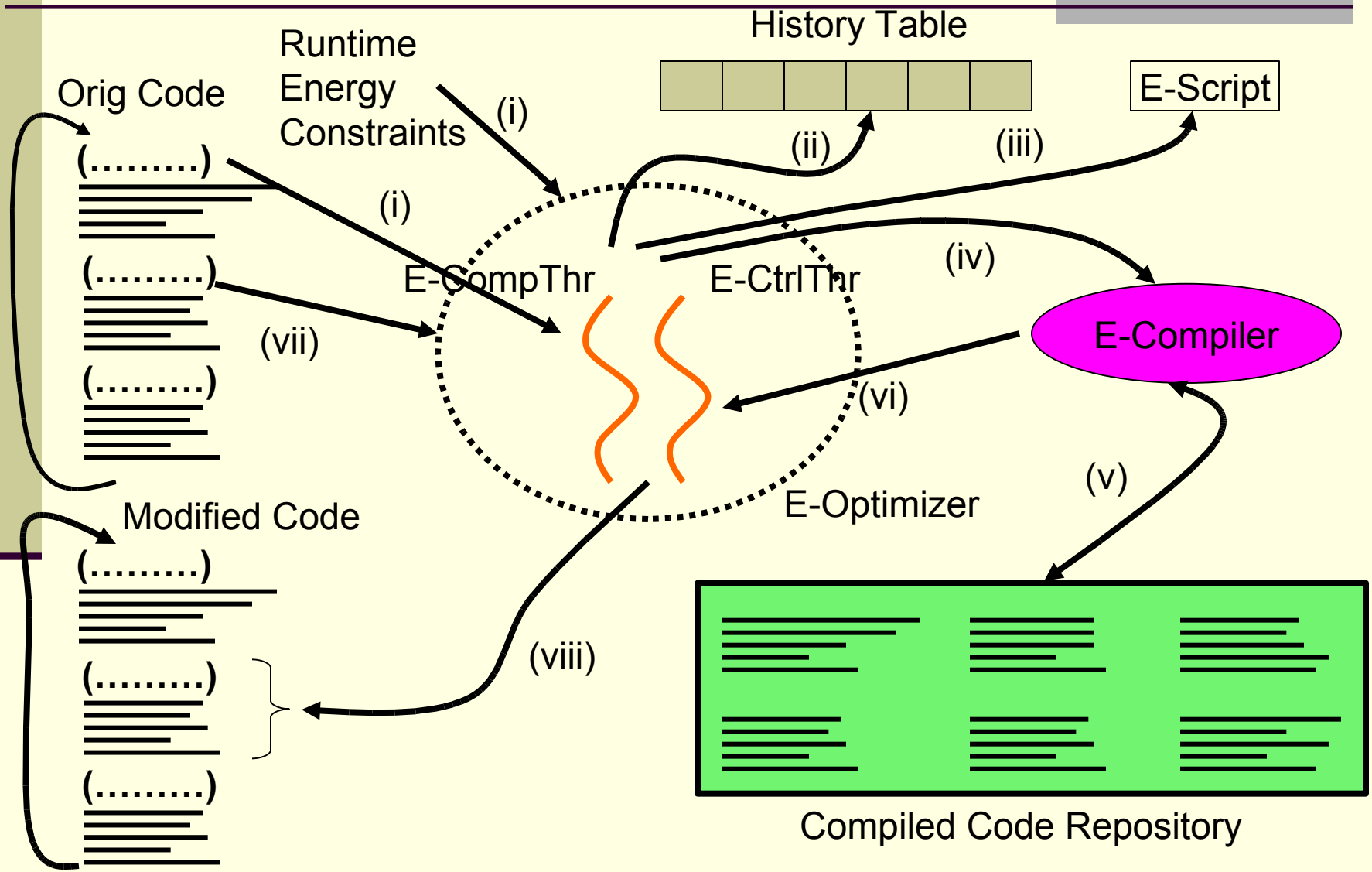
Implementation

- Dynamic compilation infrastructure
  - Dyninst software: a post-compiler program manipulation tool
  - Instrument and modify application programs during execution
  - Through Dyninst API, a program can create new code pieces and insert them to another program while the latter is executing
- Target environment
  - An energy-sensitive SoC platform with multiple cores
  - Dynamic compilation invoked due to changes in energy constraints

Abstraction used in Dyninst API



Execution Model



History-based Next Module Prediction

- Regularity within the function/subroutine/nest traces of applications
  - Array-based application: loop nests
  - Others: function/subroutine
- Reasonable prediction accuracy
- History table
  - Once an energy-sensitive region reached, it is logged into history table
  - E-CompThr consults history table before predicting the next energy sensitive region

Next Module Prediction Accuracy

Benchmark	Source/Type	Size/Input File
btrix	Spec95	721KB
tomcatv	Spec95	836KB
vpenta	Spec95	770KB
hier	Motion Est.	310KB
full_search	Motion Est.	310KB
epic	MediaBench	test_image.pgm
rasta	MediaBench	ex5_c1.wav
181.mcf	Spec2000	inp.in

Benchmark	Prediction Accuracy
btrix	99.95%
tomcatv	88.38%
vpenta	99.96%
hier	100.00%
full_search	100.00%
epic	91.02%
rasta	72.13%
181.mcf	87.52%

Time Contribution of Dynamic Compilation

- Worst/Best case time contribution of dynamic compilation

Benchmark	Compilation Time (Worst Case)	Compilation Time (Best Case)	Execution Time (msec)
btrix	17651.69 [53.1%]	1810.51 [10.4%]	15599.96
tomcatv	22225.82 [80.3%]	1854. [25.4	5450.17
vpenta	20060.68 [36.5%]	1834	4.98
hier	3515.73 [36.7%]	13	6053.28
full_search	2468.43 [70.6%]		1029.84
epic	90476.59 [82.0%]	1	19878.16
rasta	85515.07 [86.4%]	3138. [18.9%]	13436.02
181.mcf	5283755.26 [85.8%]	1490.7 [0.2%]	873999.61

Best Case: 13% on average

Experimental Results

- Various compilation probabilities
  - 100%: every module has to be recompiled before it can be executed
  - 50%(25%): the recompilation probabilities only 50% (25%) for each module
  - Most applications have compilation probabilities ranging from 25% to 80%
- Sun multi-processor machine running Solaris
  - Processor 1: application process
  - Processor 2: E-Optimizer (E-CtrlThr and E-CompThr)

Reductions in Overall Execution Time

Benchmark	2 Processors			3 Processors		
	Compl%	Excn%	Total%	Compl%	Exec%	Total%
btrix	23.11	-4.16	10.26	20.07	-0.24	10.50
tomcatv	30.80	-90.13	6.63	14.02	-5.32	10.15
vpenta	40.25	-2.38	13.17	38.69	-1.75	13.00
hier	63.75	-0.77	22.92	61.76	-0.68	22.25
full_search	27.06	-3.96	17.94	22.98	-2.28	15.55
epic	13.72	-39.36	4.01	28.66	-7.35	22.07
rasta	28.32	-162.50	2.00	36.71	-11.85	30.01
181.mcf	29.49	-129.02	5.93	38.25	-7.04	31.51

Benchmark	2 Processors			3 Processors		
	Compl%	Excn%	Total%	Compl%	Excn%	Total%
btrix	22.28	-0.67	8.34	15.99	0.38	5.65
tomcatv	8.48	-26.21	-2.25	9.92	-3.77	5.07
181.mcf	9.05	-28.50	-0.34	10.70	-9.65	4.84
vpenta	37.32	-0.47	5.89	63.42	-0.27	6.31
hier	37.27	0.43	9.22	60.86	0.32	9.27
full_search	-40.03	-4.85	-17.72	-9.55	-0.85	-4.40
epic	14.95	-11.19	2.87	8.87	-4.79	2.16
rasta	1.56	-49.92	-17.68	6.01	-6.32	1.19
181.mcf	-0.12	-10.11	-4.48	-8.59	1.08	-4.84

2 processors:
Compilation probability 100%(50%): 10.35%(4.07%) reduction
Compilation probability 25%: 2.19% increase

3 processors:
Processor 2 (E-CtrlThr); Processor 3 (E-CompThr)
Compilation probability 100%(50%): 19.38%(7.05%) reduction
Compilation probability 25%: 0.81% increase

Impact of Compiling Critical Modules

- Critical module: the one that contributes to the overall execution time significantly

Nests	Compile(ms)	Total(ms)
None	0	54375.86
Nest I	2874.97	55963.48
Nest I + Nest II	4824.32	58135.06
Nest I + Nest II + Nest IV	7319.83	46720.90
Nest I + Nest II + Nest IV + Next VIII	9692.83	59210.55
All Nests	20060.67	54989.12

Compiling only critical modules when interleaving execution
And compilation: 14.07% in total time

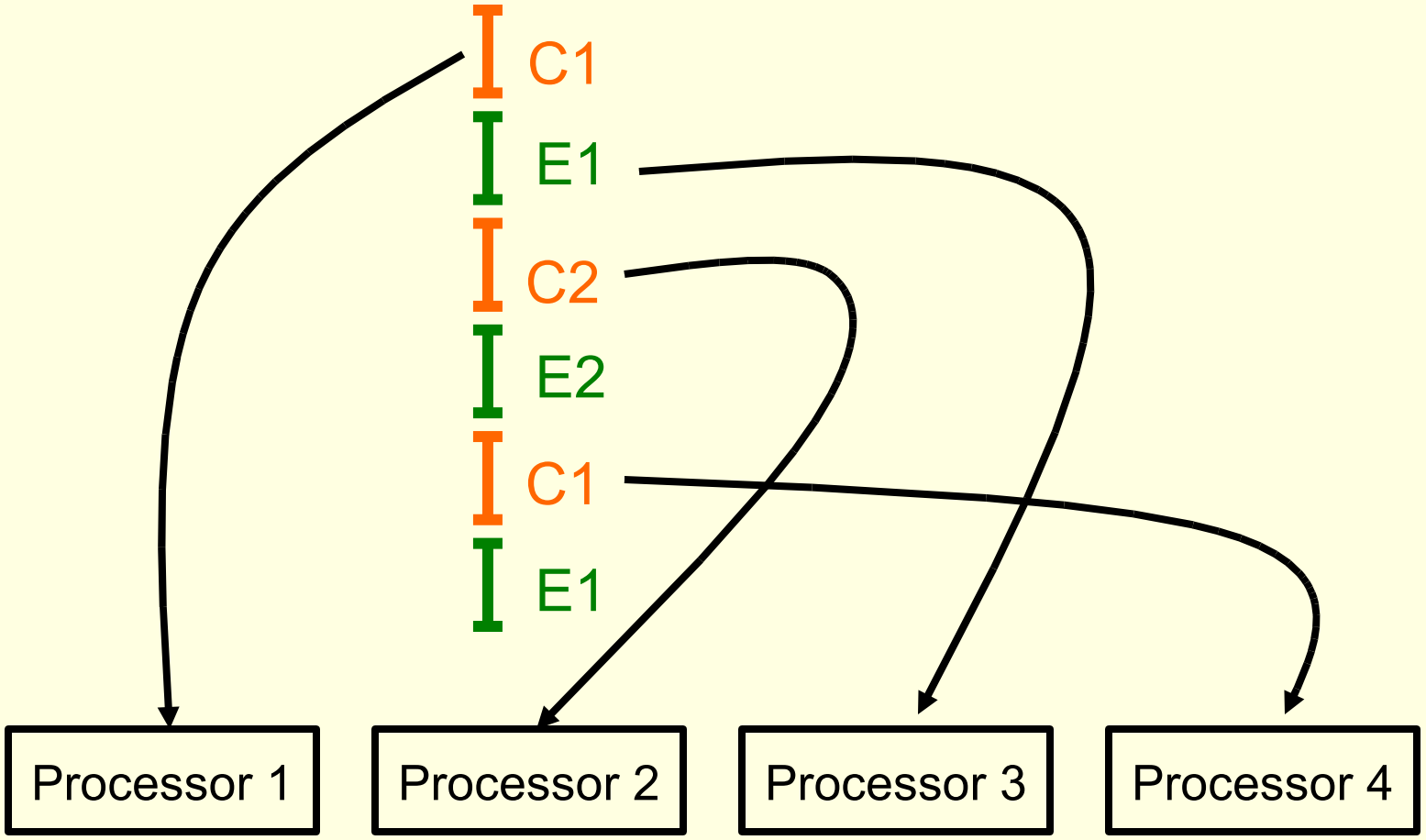
Impact of Compiling Critical Modules

Nests	Compile (msec)	Total (msec)
Nest I	262.98 [90.85%]	52614.01 [5.99%]
Nest I + Nest II	472.14 [90.21%]	53585.28 [7.83%]
Nest I + Nest II + Nest IV	2652.98 [63.76%]	39124.29 [10.58%]
Nest I + Nest II + Nest IV + Next VIII	3169.78 [67.30%]	52648.76 [11.08%]

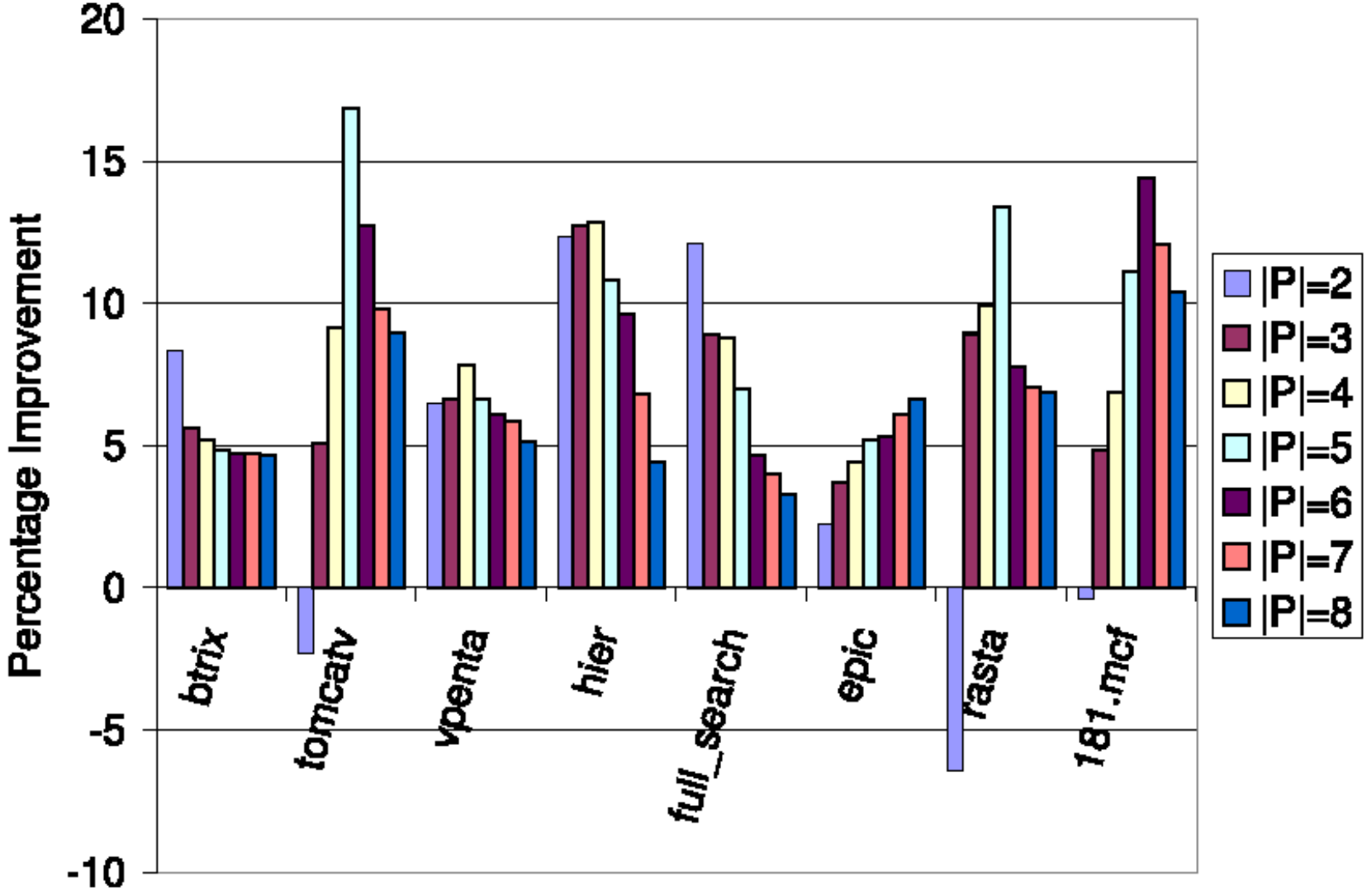
Overlapping compilation and computation: additional 13.98%
Total reduction: 28.05%

- Profile the application to determine the critical modules
- Pre-compile these modules using the proposed approach

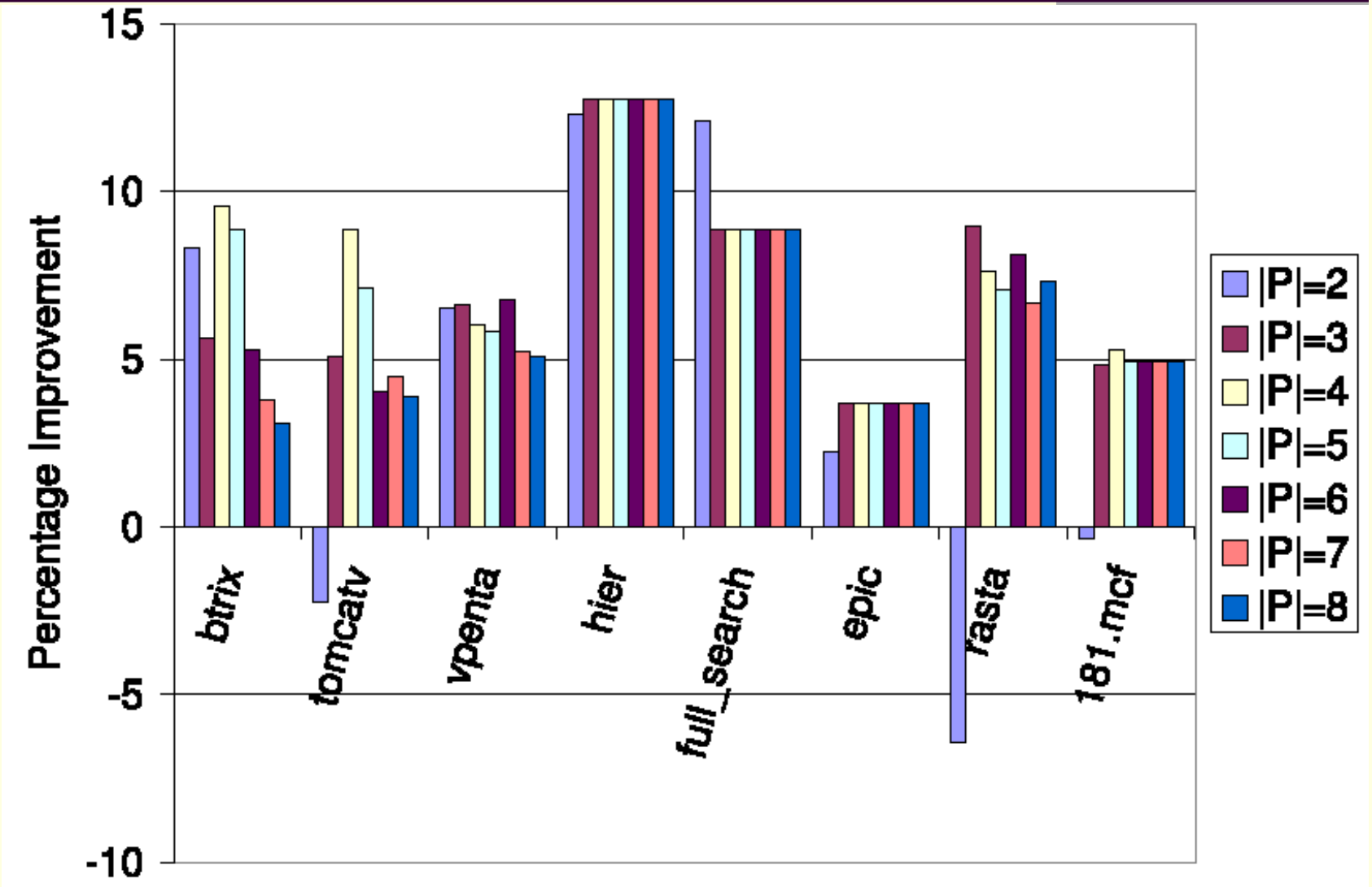
Impact of Increasing the Number of Processors



Increasing the Number of Processors for Dynamic Compilation



Increasing the Number of Processors for Application Execution



Conclusion

- Hide the time spent in dynamic compilation by overlapping it with application execution
- Implement a dynamic compilation/linking infrastructure that compiles/links program modules based on external energy constraints
  - Predicting and pre-compiling
- Encouraging experimental results

Thank you!