Dynamic Compilation: The Benefits of Early Investing

Prasad Kulkarni, FSU, IBM
Matthew Arnold IBM
Michael Hind, IBM

Introduction

- Java Virtual Machine (JVM)
  - load *bytecodes* as input
  - initial execution by interpretation or quick compilation
  - method optimized, if *hot*
- Adaptive optimization system
  - monitor running application
  - detect hot methods for further optimization

When to Compile ?

- Separate thread for JIT compilation
  - provides isolation between compiler & application
  - compiler can execute asynchronously
- Scheduling JIT compiler thread
  - round-robin scheduling considered most fair
- Processor utilization for the compiler thread
  - low priority optional VM component
  - background thread, no interference with application
- *Are traditional implementations of asynchronous JIT compilations optimal ?*

How to Exploit Free Cycles ?

- Modern machines can have un-utilized computational resources
  - multi-core or multi-processor machines
- Adapt compiler strategy
  - compilation is *free*
  - compile more aggressively
- *How to adapt compilation strategy to exploit free cycles ?*

Outline

- Experimental Setup
- Compiler Scheduling and Utilization
- Exploiting Free Processor Cycles
- Conclusions

Experimental Setup

- Used IBM's J9 VM
  - includes a high-performance JIT compiler
  - uses counters and sampling to promote methods for compilation
  - compilation performed on a *single* separate thread
- Three configurations of Intel Xeon 2.8GHz processors, Red Hat Linux kernel 2.6.9
  - single-processor
  - single-processor with hyperthreading
  - two-processors with hyperthreading

Benchmarks

- Benchmark suite with 23 programs
  - complete SPECjvm98 suite of 7 benchmarks
  - complete DaCapo suite of 10 benchmarks
  - SPECjbb2000 benchmark
  - 5 other benchmarks
    - daikon, kawa, ipsixql, soot, and xerces
- IBM Trade Performance Benchmark V6.1
  - over 40,000 methods, over 6,000 compilations

Outline

- Experimental Setup
- Compiler Scheduling and Utilization
- Exploiting Free Processor Cycles
- Conclusions

Limitations of Asynchronous Compilation Implementations

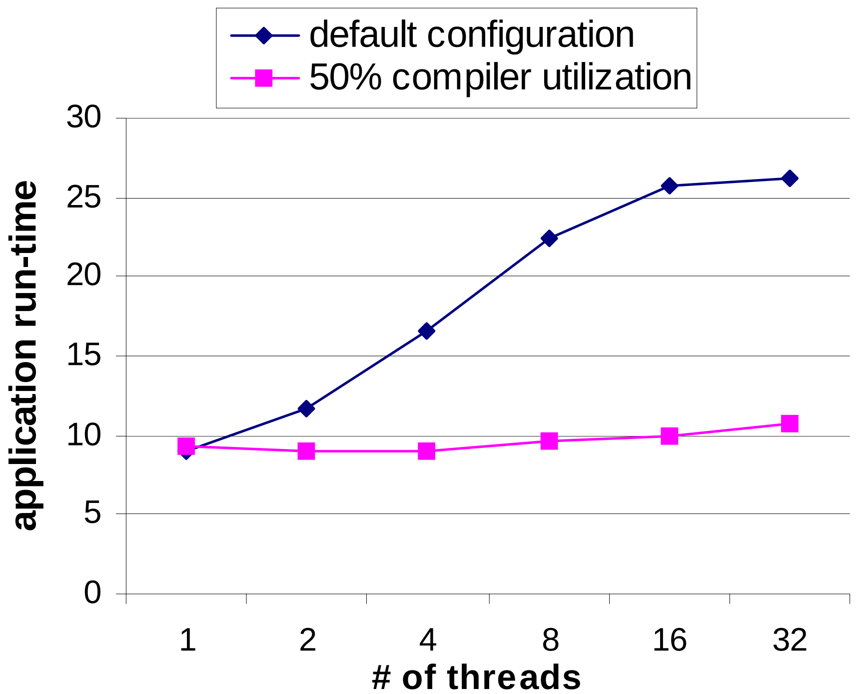
- Traditional asynchronous compilation
  - execute compiler in a separate thread
  - round-robin scheduling of threads
  - large OS scheduling times to minimize overhead
    - 100 msec time quantum, 400 million cycles/time-slice
- Reduced resources due to multi-threading
  - with *N* application threads, compilation thread resources are *1/(N+1)*
- Reduced resources due to yielding
  - I/O, message passing, etc.
  - compiles can be delayed for a long time

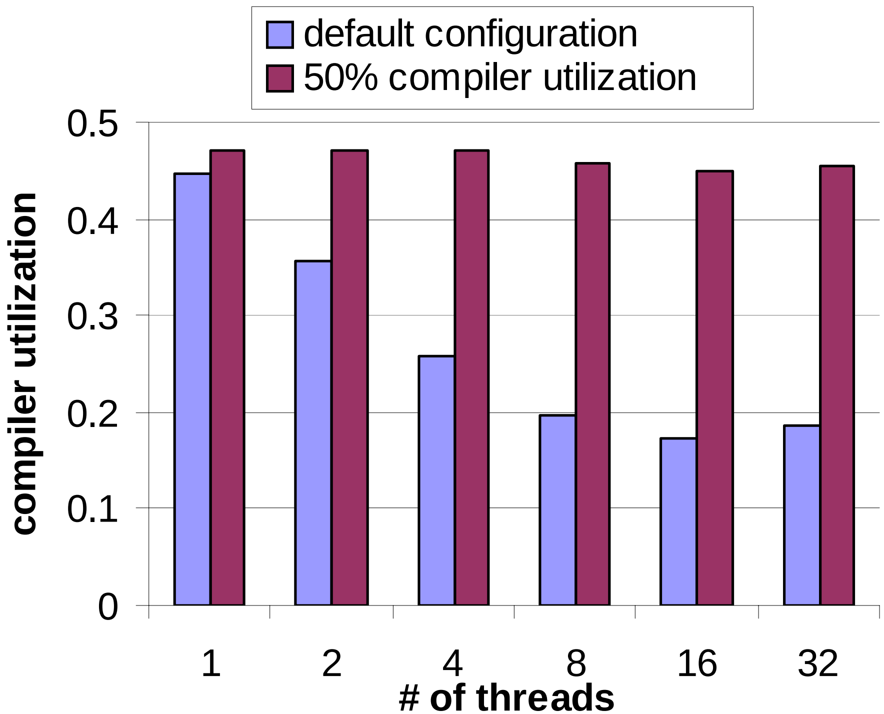
Utilization-Based Scheduling

- Construct a scheduler to enforce a specific compiler thread utilization
  - compiler thread receives exactly *X%* of CPU resources
- Used *pthread* priorities in the Linux OS
  - scales well to multi-processor machines
  - simplifies the scheduler implementation
- Define a VM-level time-slice quantum of 10ms

Effect of Round-Robin Scheduling

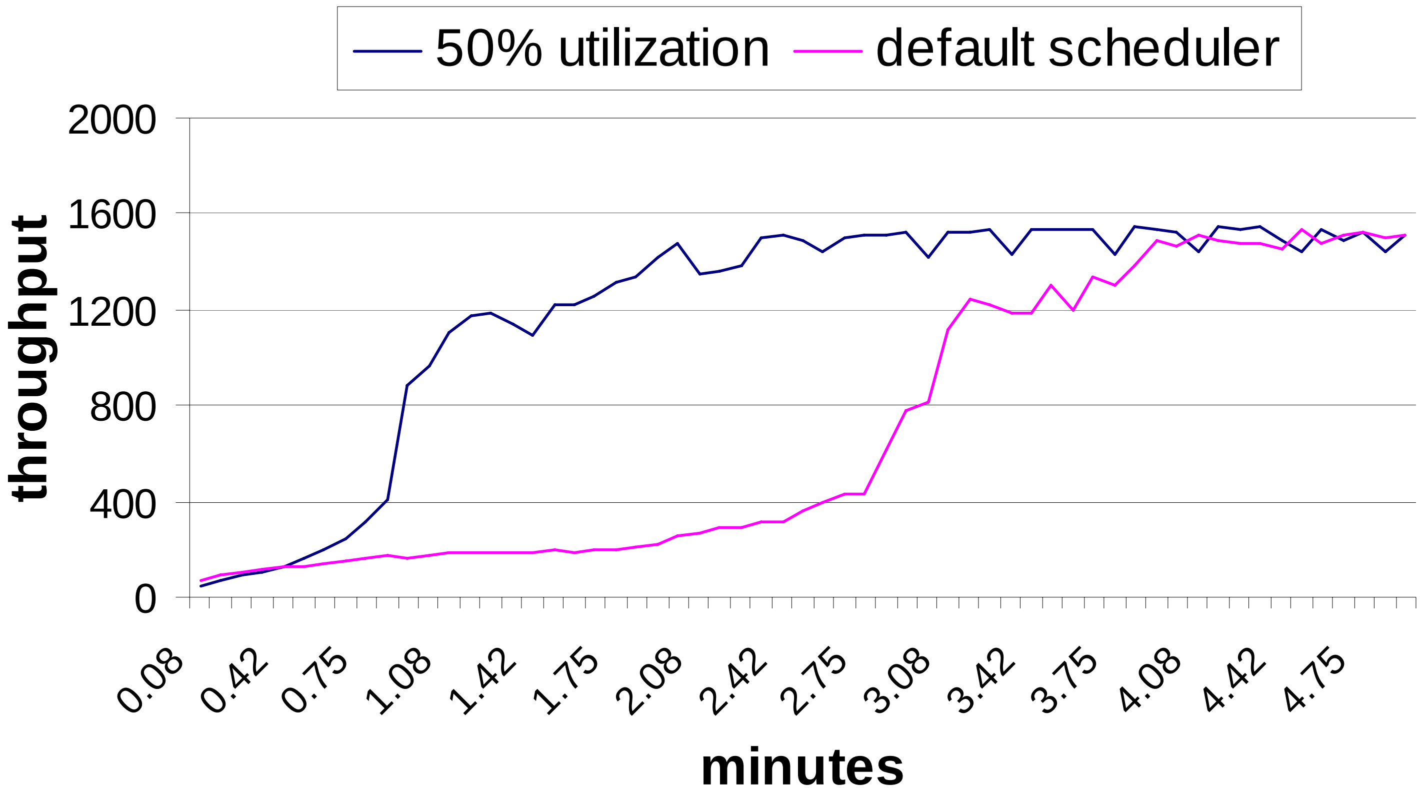
- Effect of increasing number of app. threads for *mtrt* from SPECjvm98





Effect of Round-Robin Scheduling (cont…)

- Effect on the rampup-time of Trade 6.1.



Selecting Compiler Thread Utilization

- Determine the best compiler utilization
- Evaluate 2 controller policies
  - aggressive controller
    - first compile is at optimization level O1
    - optimized for reaching steady-state quickly
  - conservative controller
    - first compiler is at optimization level O0
    - optimized for better startup performance

Compiler Thread Utilization – Aggressive Controller

	Compiler Thread Utilization										Default Scheduler
	10%	*20%*	*30%*	*40%*	*50%*	*60%*	*70%*	*80%*	*90%*	*100%*	
Perf. Imp. (%)	-85.8	-39.2	-17.0	-4.3	2.2	8.3	11.7	14.8	16.9	18.2	0.0
Time in queue (ms)	6500	3885	2408	1412	956	509	254	96	32	21	1465
Length of queue	157	112	90	61	46	27	14	5	2	1	60
Methods compiled	399	472	516	550	572	585	584	569	551	523	567

- 18.2% performance gain at 100% utilization.
- Small improvement at 50%, since most benchmarks are single-threaded
- Performance degrades as compiler utilization is reduced

Compiler Thread Utilization – Aggressive Controller

	Compiler Thread Utilization										Default Scheduler
	10%	*20%*	*30%*	*40%*	*50%*	*60%*	*70%*	*80%*	*90%*	*100%*	
Perf. Imp. (%)	-85.8	-39.2	-17.0	-4.3	2.2	8.3	11.7	14.8	16.9	18.2	0.0
Time in queue (ms)	6500	3885	2408	1412	956	509	254	96	32	21	1465
Length of queue	157	112	90	61	46	27	14	5	2	1	60
Methods compiled	399	472	516	550	572	585	584	569	551	523	567

- Time between scheduling and compilation of each method
- Queue delay progressively reduced as utilization is increased

Compiler Thread Utilization – Aggressive Controller

	Compiler Thread Utilization										Default Scheduler
	10%	*20%*	*30%*	*40%*	*50%*	*60%*	*70%*	*80%*	*90%*	*100%*	
Perf. Imp. (%)	-85.8	-39.2	-17.0	-4.3	2.2	8.3	11.7	14.8	16.9	18.2	0.0
Time in queue (ms)	6500	3885	2408	1412	956	509	254	96	32	21	1465
Length of queue	157	112	90	61	46	27	14	5	2	1	60
Methods compiled	399	472	516	550	572	585	584	569	551	523	567

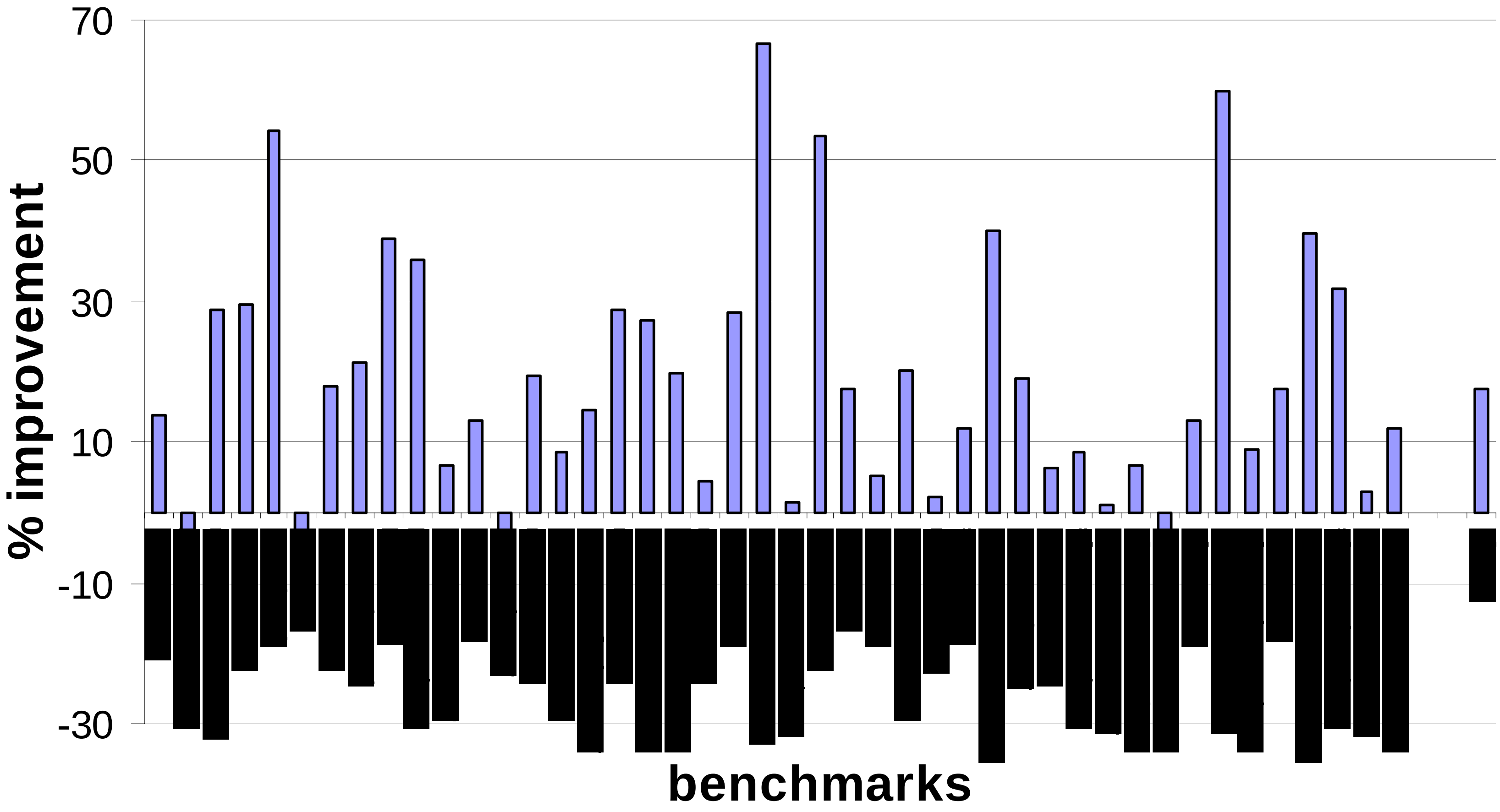
- Average number of methods in the compilation queue
- Reduces with increase in compiler utilization

Compiler Thread Utilization – Aggressive Controller

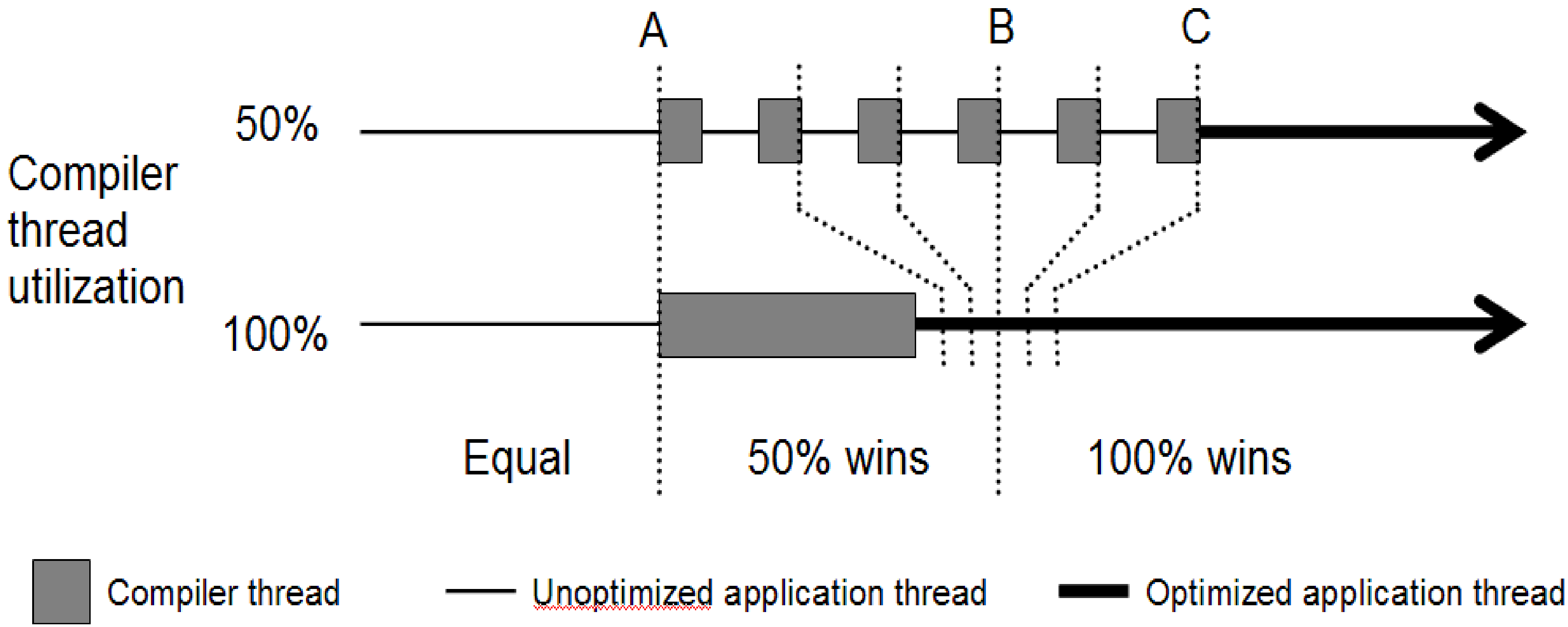
	Compiler Thread Utilization										Default Scheduler
	10%	*20%*	*30%*	*40%*	*50%*	*60%*	*70%*	*80%*	*90%*	*100%*	
Perf. Imp. (%)	-85.8	-39.2	-17.0	-4.3	2.2	8.3	11.7	14.8	16.9	18.2	0.0
Time in queue (ms)	6500	3885	2408	1412	956	509	254	96	32	21	1465
Length of queue	157	112	90	61	46	27	14	5	2	1	60
Methods compiled	399	472	516	550	572	585	584	569	551	523	567

- Number of methods compiled during the application's execution
- Compilations decrease at high utilizations because of reduced execution time

Performance Improvement at 100% Utilization



High Compiler Thread Utilization



- If the controller is making good compilation decisions, then high compiler thread utilization should be better.

Compiler Thread Utilization – Conservative Controller

	Compiler Thread Utilization										Default Scheduler
	10%	*20%*	*30%*	*40%*	*50%*	*60%*	*70%*	*80%*	*90%*	*100%*	
Perf. Imp. (%)	-45.7	-10.7	0.2	7.0	9.7	10.2	10.1	9.9	9.5	9.3	0.0
Time in queue (ms)	862	301	142	54	30	16	9	3	1	1	130
Length of queue	83	41	21	12	7	4	3	2	1	1	25
Methods compiled	632	682	688	696	702	694	690	684	678	672	705

- More methods compiled
- Smaller performance improvements
- Insignificant backup of methods in queue

Compiler Thread Utilization – Conservative Controller

	Compiler Thread Utilization										Default Scheduler
	10%	*20%*	*30%*	*40%*	*50%*	*60%*	*70%*	*80%*	*90%*	*100%*	
Perf. Imp. (%)	-45.7	-10.7	0.2	7.0	9.7	10.2	10.1	9.9	9.5	9.3	0.0
Time in queue (ms)	862	301	142	54	30	16	9	3	1	1	130
Length of queue	83	41	21	12	7	4	3	2	1	1	25
Methods compiled	632	682	688	696	702	694	690	684	678	672	705

- 50%-90% utilizations slightly better than 100% utilization
- Improvements here seem to be guided more by latency than utilization

Impact on Pause Times

- Pause time considerations
  - insignificant for batch applications
  - important for applications with real-time constraints
- 100% utilization cannot provide strong pause time guarantees
- 60-80% utilization can provide higher performance and better pause time guarantees than round-robin scheduler

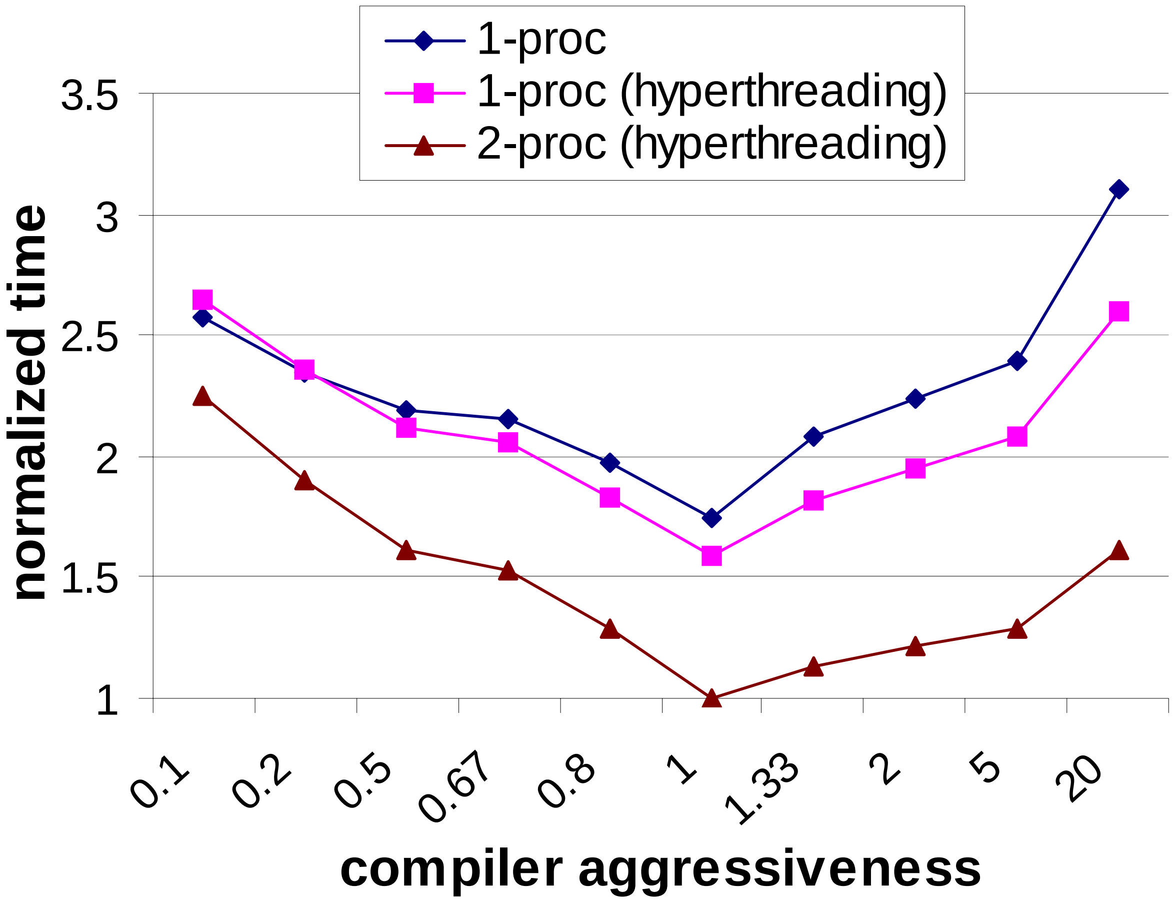
Outline

- Experimental Setup
- Compiler Scheduling and Utilization
- Exploiting Free Processor Cycles
- Conclusions

Issues for Multiprocessor Machines

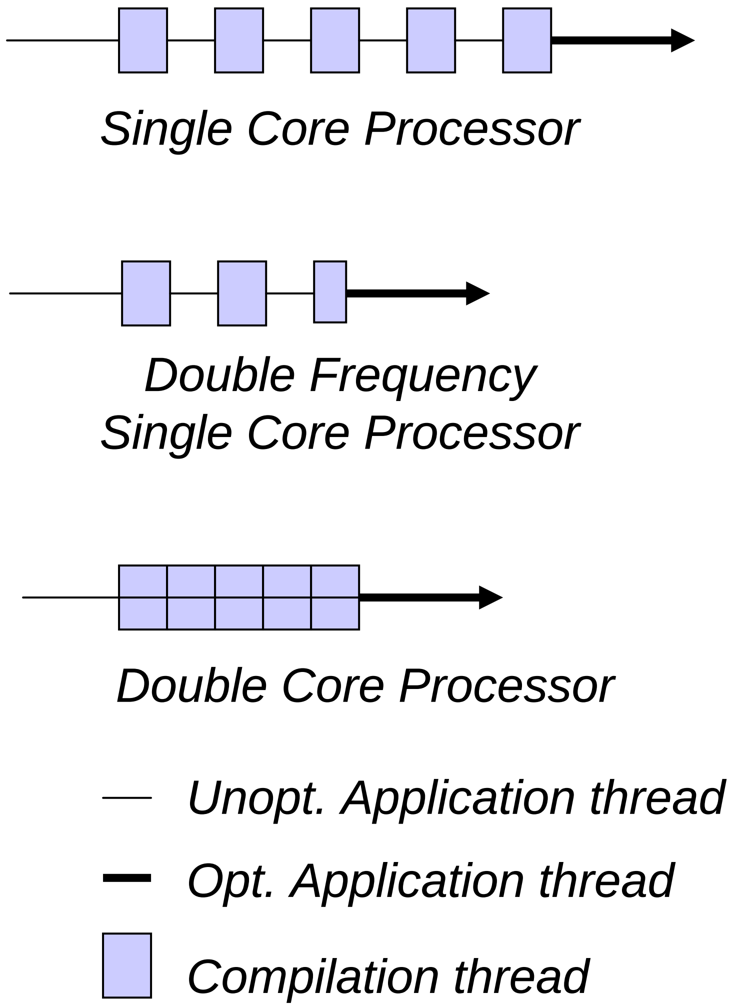
- Higher chances of unused processor resources
  - spread of multi-core machines
  - unable to exploit more parallelism
- Adapt controller policy to exploit *free cycles*?
  - compile more aggressively
- Simple motivating experiment
  - tune compilation aggression on different machines
  - demonstrate that no single strategy works on all machines
    - more aggressive strategy needed to exploit free processor

Changing Compiler Aggressiveness



Analyzing Aggressive Compilation

- On a multiprocessor machine with 1 application and 1 compiler thread
  - both threads are making equal progress relative to each other
  - similar to doubling frequency of single processor machine



Analyzing Aggressive Compilation (cont…)

- Compiling more aggressively
  - introduces additional *secondary* compiles
  - increases length of compilation queue
  - delays *primary* compiles
- *If ratio of application to compiler threads is unchanged, then compiler strategy should not be changed.*

Exploiting Free Cycles

- Schedule secondary compiles when
  - no primary compiles left
  - idle processor cycles available
  - can preempt secondary compiles for primary compile
- Strategy did not result in significant performance benefit on our system.

- *Little incentive to change compilation strategy for single compiler thread VM.*

Multiple Compilation Threads

- Seem to be the right approach for multiprocessor machines
  - can effectively exploit free cycles
  - compiling early has been shown to benefit performance
- Processor utilization for compilation can be controlled
  - imposing a utilization for each processor
  - spawning variable number of compiler threads

Outline

- Experimental Setup
- Compiler Scheduling and Utilization
- Exploiting Free Processor Cycles
- Conclusions

Conclusions

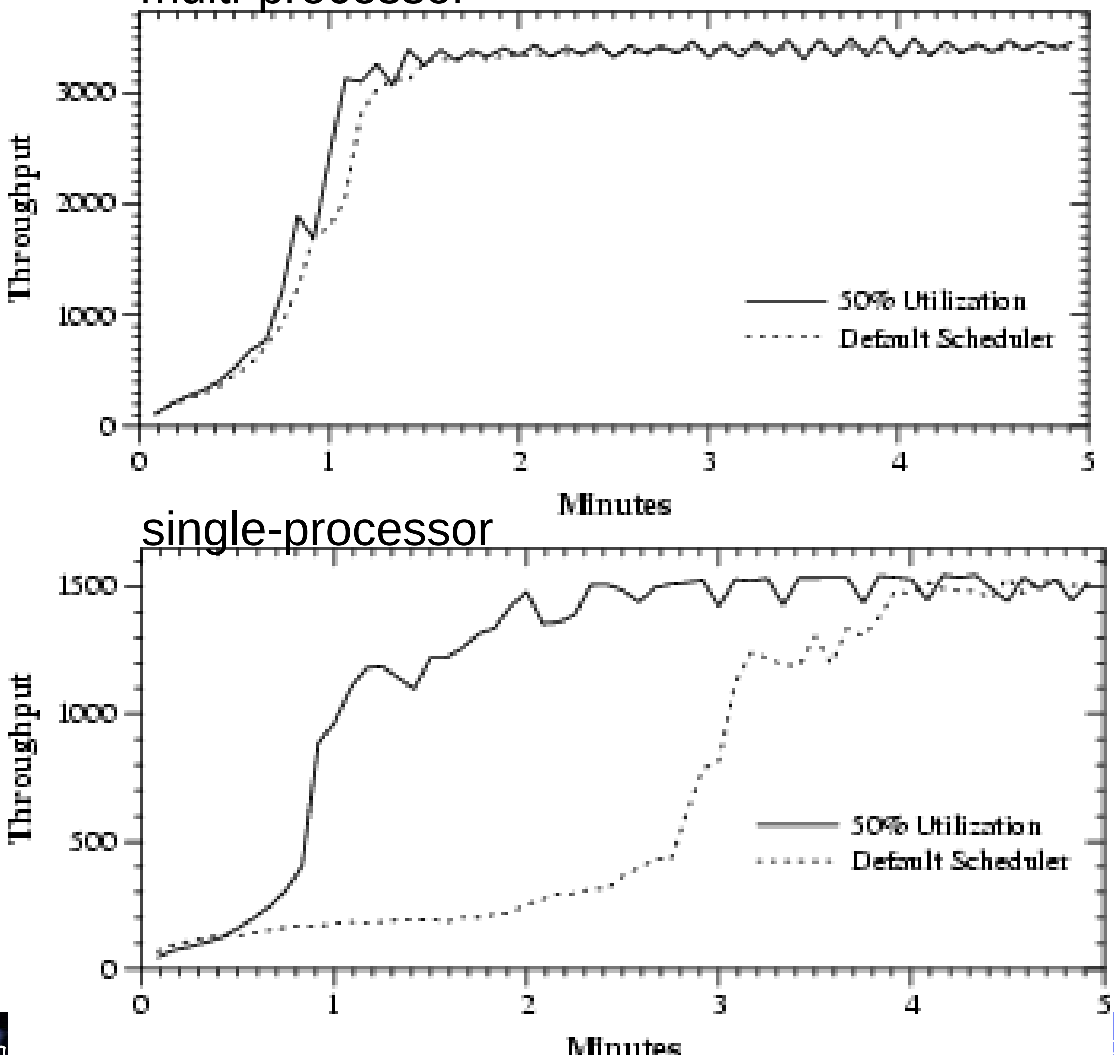
- It is necessary to guaranty a certain level of utilization for the compiler thread.
- Higher compiler utilization result in good performance for tuned controllers
  - 18% speedup for aggressive controller
  - 9% speedup for conservative controller
- Controller policy for single compiler thread does not need to change for multiprocessor machines.

Effect of Round-Robin Scheduling (cont…)

- Effect on the rampup-time of Trade 6.1.

multi-processor

single-processor



Background Compilation

- Compiler is executed in a separate thread
- Advantages
  - provides isolation between the run-time states of the compiler and application
  - compiler can execute asynchronously with application threads
- Implementation
  - OS threads with round-robin scheduling
  - VM threads multiplexed over OS threads

Impact on Pause Times

Window (ms)	Specified Compilation Thread Utilization										RR
	10%	*20%*	*30%*	*40%*	*50%*	*60%*	*70%*	*80%*	*90%*	*100%*	
100	77	56	51	45	46	31	24	14	6	0	1
300	84	72	62	55	49	37	28	19	10	1	24
600	87	77	66	58	49	39	30	22	16	13	40
1000	88	79	68	59	50	41	33	26	22	20	45

- MMU: minimum application utilization during some time-parameterized window
- Benchmark with the worst MMU determined the value reported

Impact on Pause Times

Window (ms)	Specified Compilation Thread Utilization										RR
	10%	*20%*	*30%*	*40%*	*50%*	*60%*	*70%*	*80%*	*90%*	*100%*	
100	77	56	51	45	46	31	24	14	6	0	1
300	84	72	62	55	49	37	28	19	10	1	24
600	87	77	66	58	49	39	30	22	16	13	40
1000	88	79	68	59	50	41	33	26	22	20	45

- 100% utilization cannot provide strong MMU guarantees

Impact on Pause Times

Window (ms)	Specified Compilation Thread Utilization										RR
	10%	*20%*	*30%*	*40%*	*50%*	*60%*	*70%*	*80%*	*90%*	*100%*	
100	77	56	51	45	46	31	24	14	6	0	1
300	84	72	62	55	49	37	28	19	10	1	24
600	87	77	66	58	49	39	30	22	16	13	40
1000	88	79	68	59	50	41	33	26	22	20	45

- 100% utilization cannot provide strong MMU guarantees
- Lower utilization provides better performance and MMU guarantees than round-robin scheduler

Compiler Thread Utilization - Trade