Tyler Sondag
Iowa State U.

Hridesh Rajan
Iowa State U.

Phase-guided Thread-to-core Assignment for Improved Utilization of Performance-Asymmetric Multi-Core Processors

International Workshop on Multicore Software Engineering

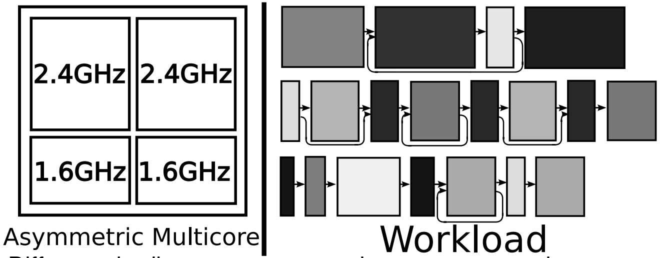
Supported in part by the US National Science Foundation under grants 06-27354 and 08-08913.

Overview

Performance asymmetric multicores are seen as a more efficient alternative to homogeneous multicores.

Broad Problem: Efficient utilization of asymmetric cores

Technical Challenge: Match resource requirements



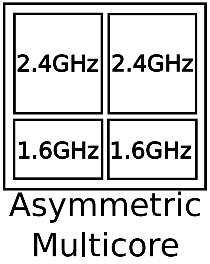
Different shading represents varying resource requirements.

- Resource needs of threads vary at runtime.
- Target architecture may not be known statically.

Key Insight: Use phase behavior to reduce runtime overhead.

Performance Asymmetric Multicores

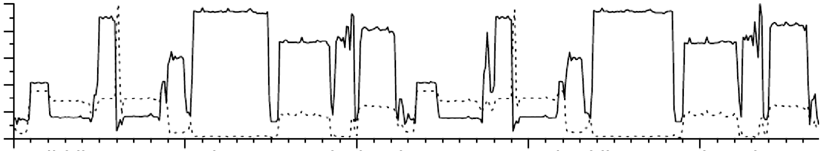
- **What**: Cores have different characteristics (clock speed, cache size, etc.)
- **Why**[1]:
  - space
  - heat
  - power
  - performance-power ratio
  - parallelism



[1] R. Kumar et al. ISCA ’04

Phase Behavior

- **Behavior:** resource requirements (IPC, cache, etc.)
- **Similar Behavior:** segments with similar resource usage
- **Phase:** segments of execution that exhibit similar behavior[2]



Solid line – Instruction Per Cycle (IPC) Dashed line – Cache misses

Phase behavior for gcc (taken from [2])

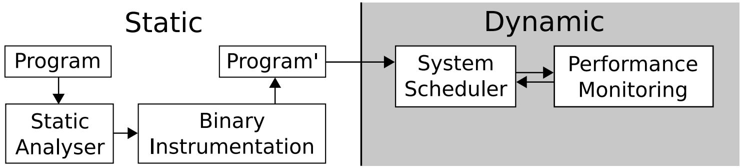
[2] T. Sherwood et al. ASPLOS '02

Intuition Behind Our Solution

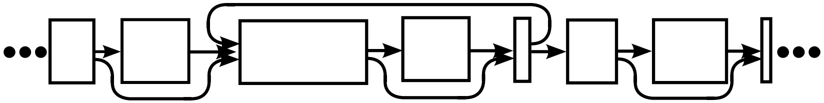
- **Problem**: Assign code to cores such that behavior of code matches resources of cores
- **Idea**:
  1. Determine sections of code that will behave in a similar way
  2. Knowledge of one section gives us information about all similar sections

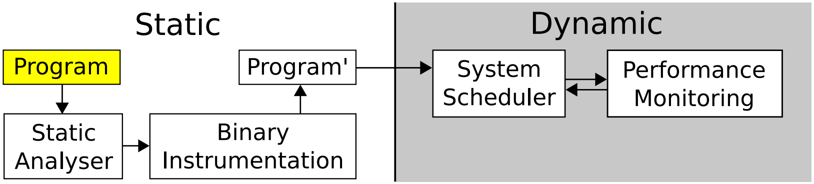
Approach Overview

- Idea: Apply the same thread-to-core mapping to all approximately similar sections of code
  1. Statically break the program into sections of code
  2. Statically determine approximate similarity between these sections
  3. Dynamically monitor a section then make mapping decisions for similar section

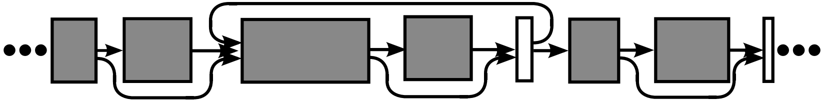


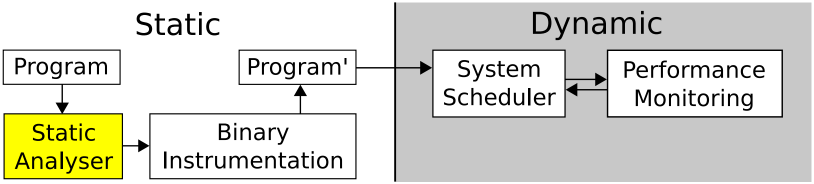
Program



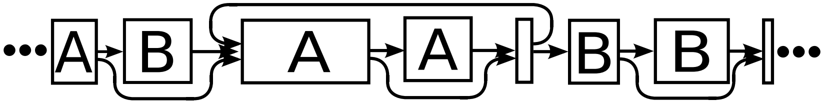

Static
Dynamic
Program
Program'
Static Analyser
Binary Instrumentation
System Scheduler
Performance Monitoring

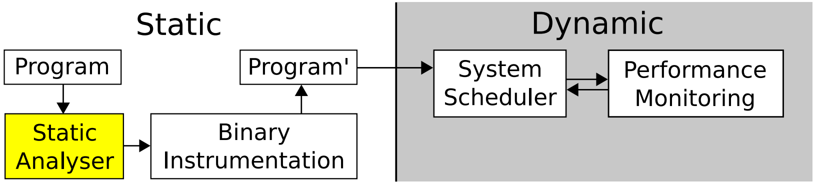
Ignore “small” sections



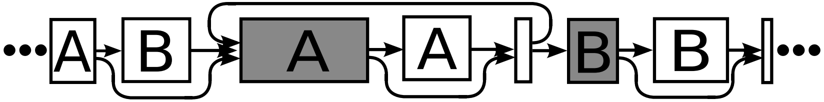


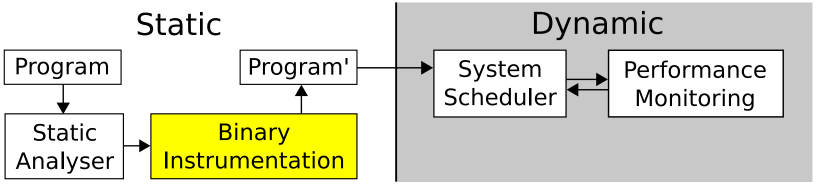
Determine approximate similarity



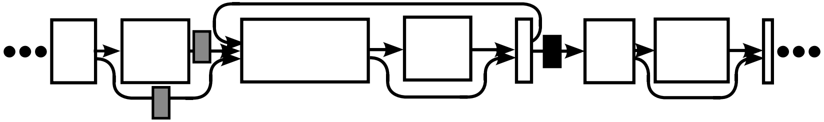


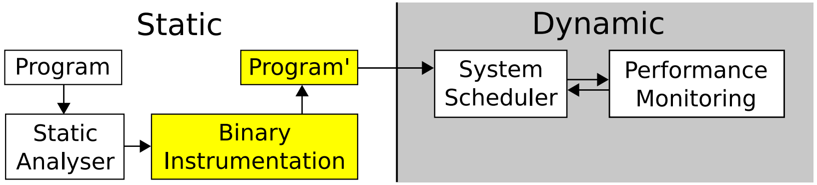
Reduce number of transition points



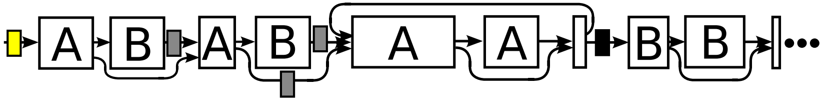


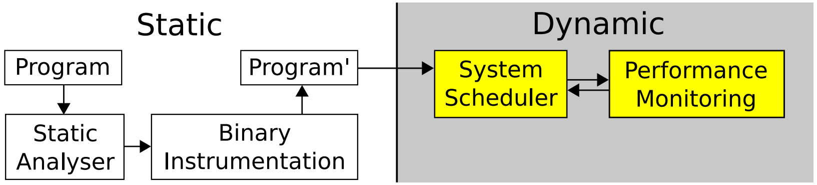
Insert phase marks



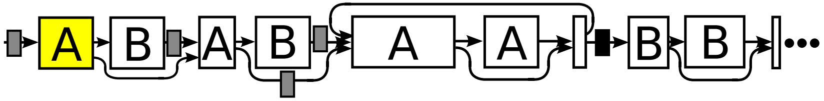


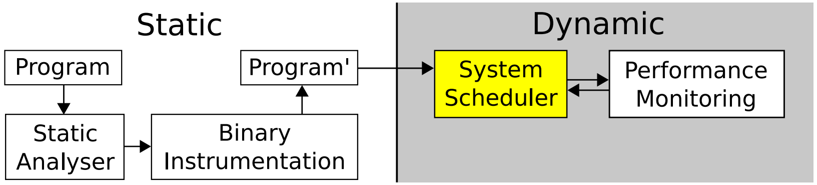
Monitor


A
B
A
B
A
A
B
B
•••

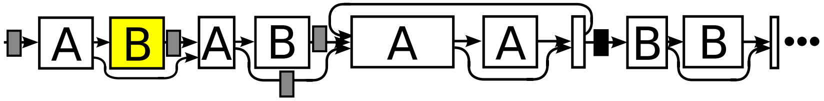

Static
Program
Static Analyser
Binary Instrumentation
Program'
Dynamic
System Scheduler
Performance Monitoring

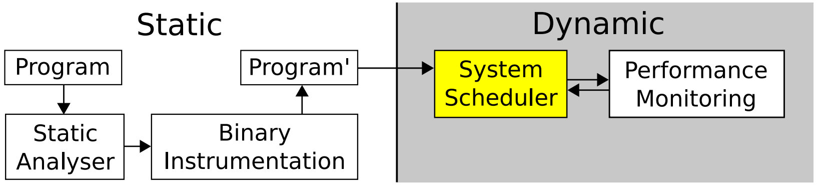
Run


A
B
A
B
A
A
B
B
•••

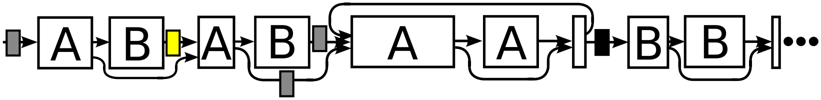

Static
Program
Static Analyser
Binary Instrumentation
Program'
Dynamic
System Scheduler
Performance Monitoring

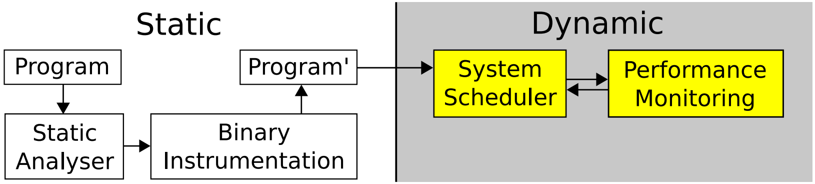
Run


A
B
A
B
A
A
B
B
•••

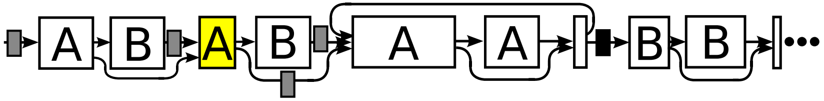

Static
Program
Static Analyser
Binary Instrumentation
Program'
Dynamic
System Scheduler
Performance Monitoring

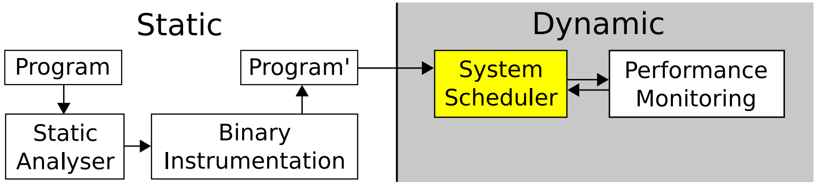
Monitor


A
B
A
B
A
A
B
B
•••

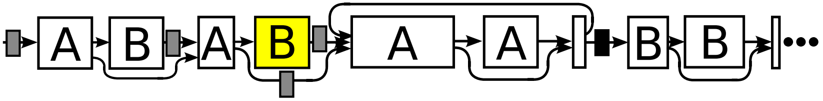

Static
Program
Static Analyser
Binary Instrumentation
Program'
Dynamic
System Scheduler
Performance Monitoring

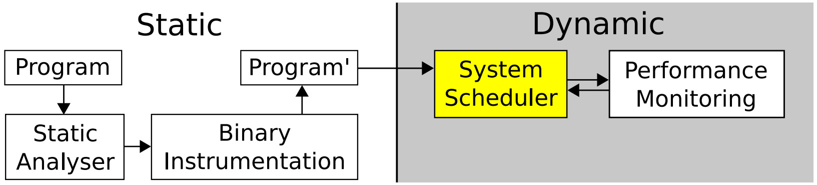
Run


A
B
A
B
A
A
B
B
•••

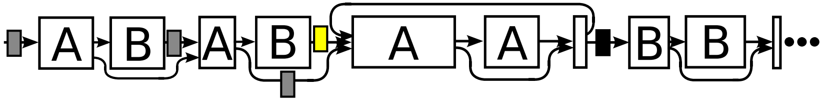

Static
Program
Static Analyser
Binary Instrumentation
Program'
Dynamic
System Scheduler
Performance Monitoring

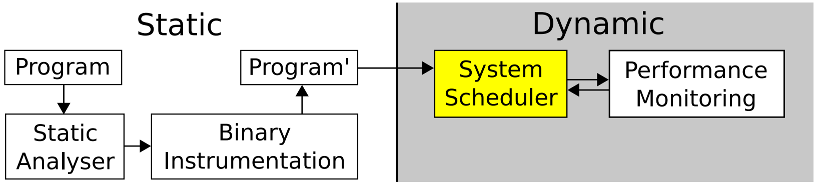
Run


A
B
A
B
A
A
B
B
•••

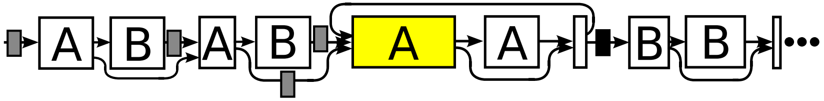

Static
Program
Static Analyser
Binary Instrumentation
Program'
Dynamic
System Scheduler
Performance Monitoring

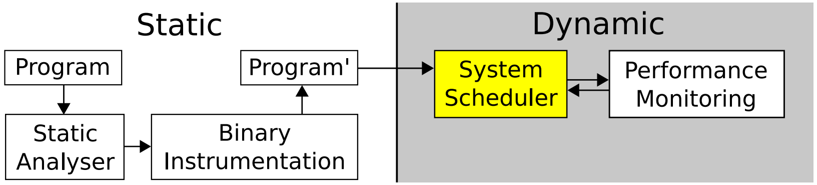
Switch to matched core





Run on matched core

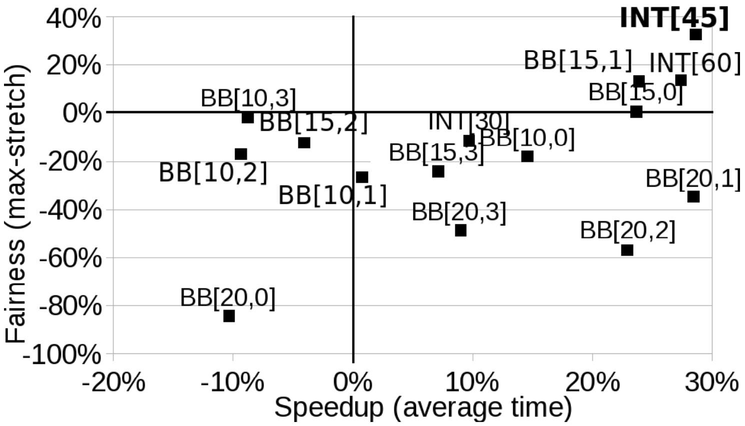




Experimental Setup

- Hardware setup: Quad Core - 2x2.4GHz, 2x1.6GHz
- Workloads
  - 36-84 SPEC CPU2000 benchmarks
  - constant workload size
- Compare to standard Linux assignment

Overall



Best Result: Interval technique, min. size 45 instructions

Previous Work

Falls into two categories

- Asymmetry-aware scheduler[3]
  - high monitoring overhead
  - requires OS modification
- Improved load balancing[4][5]
  - ignores behavior - may cause inefficient utilization
  - requires OS modification

[3] R. Kumar et al. ISCA '04
[4] T. Li et al. SC '07
[5] M. Becchi et al. CF '06

Conclusion

- Performance asymmetric multicores are a beneficial class of processors.
- Problem: Techniques to effectively assign threads to cores are still needed.
- **Solution: Use phase behavior to reduce dynamic overhead.**
  - Programmer oblivious
  - Automatic
  - Negligible overhead
  - Transparent deployment

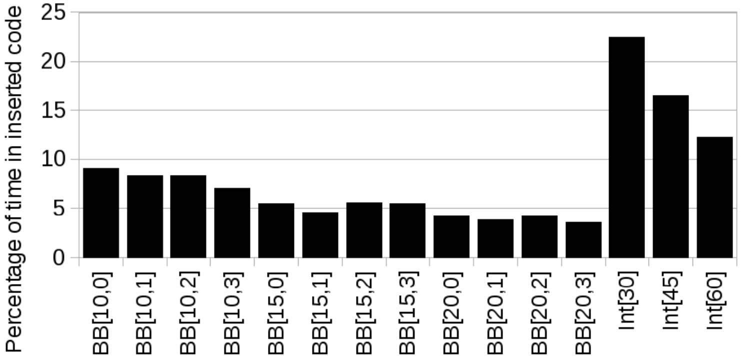
Questions

Questions?

Experimental Setup

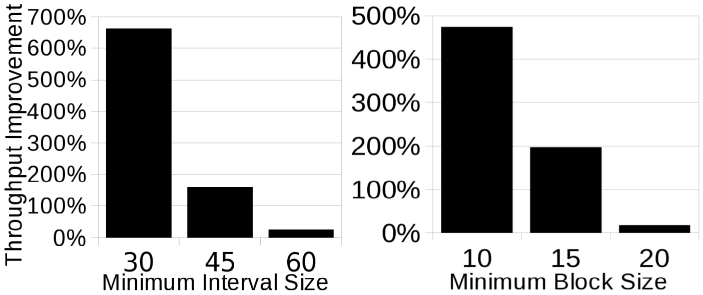
- Hardware setup: Quad Core - 2x2.4GHz, 2x1.6GHz
- Software setup
  - Static analysis/instrumentation: our framework based on GNU Binutils
  - Runtime Performance monitoring: PAPI, perfmon2
  - Core switching: affinity calls built-in to kernel
  - Workloads
    - 36-84 SPEC CPU2000 benchmarks
    - constant workload size
- Compare to standard Linux assignment

Overheads (Time)



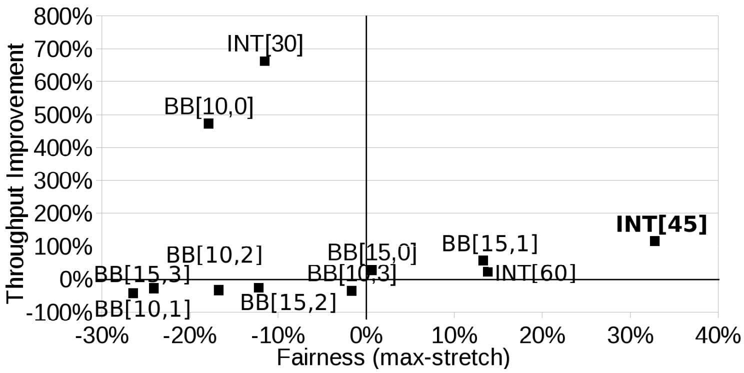
BB[x, y]: Basic block technique, min. block size: x, Look-ahead: y. **Int[x]**: interval technique, min. interval size: x

Throughput Improvement (Instructions Executed)

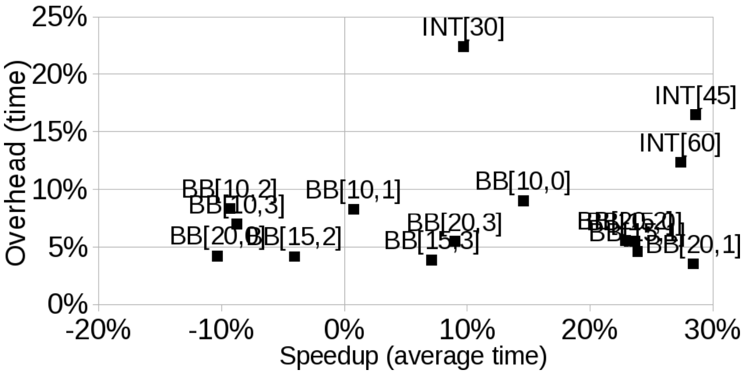


Left: Interval technique, Right: Basic block technique

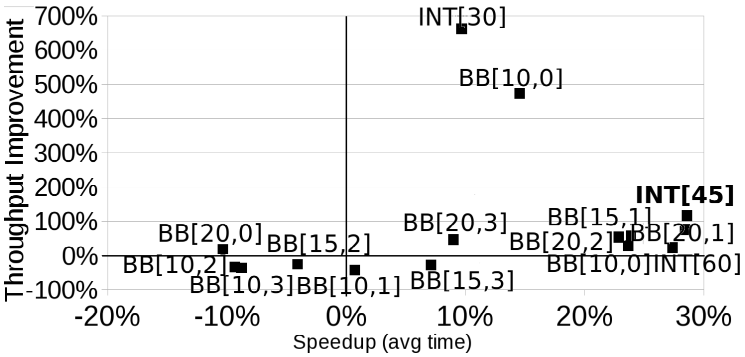
Speedup vs Fairness



Speedup vs Overhead



Speedup vs Throughput



Determining program behavior

Falls into two categories

- Techniques using execution traces
- Purely dynamic techniques

Execution Traces

- Benefits:
  - Very accurate since actual performance is known
  - Low dynamic overhead since no monitoring is required
- Limitations:
  - Requires sample input set to be developed
  - Run entire program to create execution trace
  - What about sections of code not covered by sample input?
  - Do different inputs result in different behavior?

Purely Dynamic

- Benefits:
  - Does not require sample input sets
  - No need for execution trace
  - Does not monitor the whole program
- Limitations:
  - Decisions for future code are made based on past code
  - Higher dynamic overhead since we must monitor periodically throughout the entire execution

Static Phase Marking

- Predict similarity between sections of code
- Insert phase marks on type transitions if determined beneficial
  - Basic blocks with look-ahead
  - Intervals

Monitoring and Assignment

Phase marks

- Dynamic analysis code
  - Monitor code if no mapping is unknown
  - Switch cores if mapping is known
- Type information

Asymmetry Aware Scheduler

- **What**: Scheduler assigns threads to well matched cores
- **Benefits**:
  - Very accurate since based on actual performance
  - Makes system wide decisions
  - Programs switch cores as behavior changes
- **Limitations**:
  - Monitoring is required throughout entire execution
  - Decisions for future execution are based on past behavior
  - Requires OS modification

Improved Load Balancing

- **What**: “Fast” cores get more processes or round-robin
- **Benefits**:
  - Low overhead: does not monitor execution
  - System wide decision making
- **Limitations**:
  - Aimed at fairness
  - Ignores behavior: “Fast” programs may be on “slow” cores
  - Requires OS modification

Intuition Behind Our Solution

- **Problem**: Assign code sections to cores such that behavior of code matches resources of cores
- **Idea**: If we can determine that sections of code will behave in a similar way, knowledge of one section gives us information about all similar sections.
- **Advantages** of this approach
  - Only need to monitor a small amount of code dynamically
  - No need to predict the actual behavior, just similarity.
  - Considers changes in program behavior
  - No knowledge of target machine required
  - No need for execution traces or sample inputs
  - Automatic - transparent to programmer and end user