Phase-based Tuning

for Better Utilization of Performance-Asymmetric Multicore Processors

Tyler Sondag and Hridesh Rajan

Department of Computer Science
Iowa State University

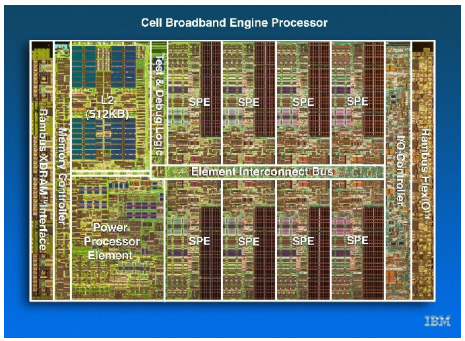
April 3, 2011

Supported in part by the US National Science Foundation under grant 00-46059.

Performance Asymmetric Multicores

- **What**: Cores may have different characteristics

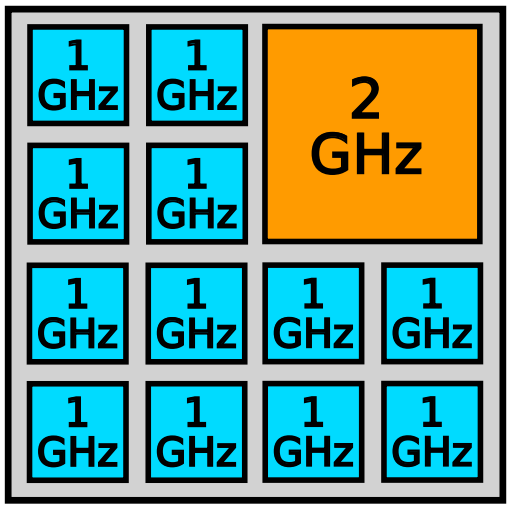
  clock speed, cache size, branch predictors, in/out-of order, etc.
- **Why** more efficient than homogeneous[1]:
  - space
  - heat
  - power
  - performance-power ratio
  - parallelism
- Asymmetry in symmetric systems.



IBM's Cell

AMD's Fusion

Intel's Sandybridge

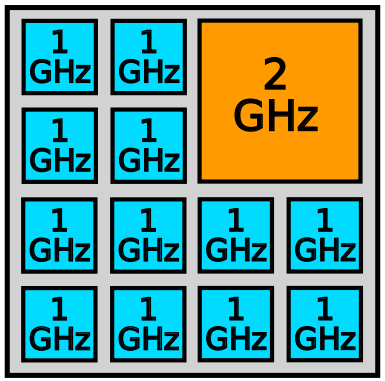


Asymmetric Multicore

[1] R. Kumar et al. ISCA '04

Overview

Problem: Efficient utilization of asymmetric cores.
Challenge: Match resource needs of threads to cores.



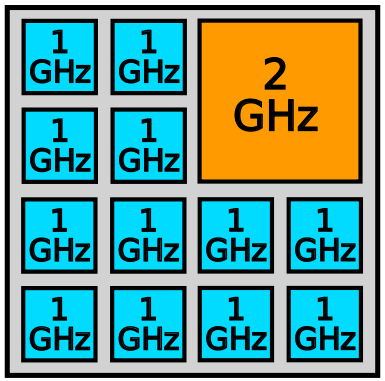
Asymmetric Multicore

```
void foo(){
  for(i=0:n)
  //cpu intense
  for(i=0:n)
  //mem intense}
```

Workload

Overview

Problem: Efficient utilization of asymmetric cores.
Challenge: Match resource needs of threads to cores.



Asymmetric Multicore

```
void foo(){
  for(i=0:n)
  //cpu intense
  for(i=0:n)
  //mem intense}
```

??

```
void foo(){
  sw(fast)
  for(i=0:n)
  //cpu intense
  sw(slow)
  for(i=0:n)
  //mem intense}
```

Workload

Overview

Technical Challenge: Matching resource requirements

Overview

Technical Challenge: Matching resource requirements

- What must be known to perform an efficient mapping?
  - Behavior of application
  - Behavior of cores

Overview

Technical Challenge: Matching resource requirements

- What must be known to perform an efficient mapping?
  - Behavior of application
  - Behavior of cores
- Can we do this manually? – tedious, error prone, expertise

Overview

Technical Challenge: Matching resource requirements

- What must be known to perform an efficient mapping?
  - Behavior of application
  - Behavior of cores
- Can we do this manually? – tedious, error prone, expertise
- statically? – unknown target, behavior changes

Overview

Technical Challenge: Matching resource requirements

- What must be known to perform an efficient mapping?
  - Behavior of application
  - Behavior of cores
- Can we do this manually? – tedious, error prone, expertise
- statically? – unknown target, behavior changes
- dynamically? – possible, but expensive

Overview

Technical Challenge: Matching resource requirements

- What must be known to perform an efficient mapping?
  - Behavior of application
  - Behavior of cores
- Can we do this manually? – tedious, error prone, expertise
- statically? – unknown target, behavior changes
- dynamically? – possible, but expensive

Insight: Use repeating behavior to reduce runtime overhead.

Contribution: Phase-based tuning

Overview

Technical Challenge: Matching resource requirements

- What must be known to perform an efficient mapping?
  - Behavior of application
  - Behavior of cores
- Can we do this manually? – tedious, error prone, expertise
- statically? – unknown target, behavior changes
- dynamically? – possible, but expensive

Insight: Use repeating behavior to reduce runtime overhead.

Contribution: Phase-based tuning

Outline

- Problem in detail
- Overview of solution
- Evaluation

Problem

Technical Challenge: Matching resource requirements

Problem

Technical Challenge: Matching resource requirements

- What must be known to perform an efficient mapping?
  - Behavior of application
  - Behavior of cores

Problem

Technical Challenge: Matching resource requirements

- What must be known to perform an efficient mapping?
  - Behavior of application
  - Behavior of cores
- Can we do this statically?
- Can we do this dynamically?

Static Solution?

Can we perform an efficient mapping statically?

Static Solution?

Can we perform an efficient mapping statically?

Difficult and frequently impossible

Static Solution?

Can we perform an efficient mapping statically?

Difficult and frequently impossible

- Manually tuning requires expertise and is error prone.
  - What is the behavior of library calls?
  - How will my code perform on each core?

Static Solution?

Can we perform an efficient mapping statically?

Difficult and frequently impossible

- Manually tuning requires expertise and is error prone.
  - What is the behavior of library calls?
  - How will my code perform on each core?
- Resource need of threads may vary at runtime.
  - Program input may change performance

Static Solution?

Can we perform an efficient mapping statically?

Difficult and frequently impossible

- Manually tuning requires expertise and is error prone.
  - What is the behavior of library calls?
  - How will my code perform on each core?
- Resource need of threads may vary at runtime.
  - Program input may change performance
- Target architecture unknown statically (multiple targets).
  - How to create a portable implementation

Static Solution?

Can we perform an efficient mapping statically?

Difficult and frequently impossible

- Manually tuning requires expertise and is error prone.
  - What is the behavior of library calls?
  - How will my code perform on each core?
- Resource need of threads may vary at runtime.
  - Program input may change performance
- Target architecture unknown statically (multiple targets).
  - How to create a portable implementation
- Resource availability may change due to contention
  - Other process on a core may demand more cache

Dynamic Solution?

- **Can we perform an efficient mapping dynamically?**

Dynamic Solution?

- **Can we perform an efficient mapping dynamically?**
- Know program behavior, cores' resources, etc.

Dynamic Solution?

- **Can we perform an efficient mapping dynamically?**
- Know program behavior, cores' resources, etc.
- Dynamically instrumentation/monitoring is expensive
  - ex: applying Pin to an application has $\geq$ 50% overhead[2]
  - Instrumentation code can introduce high overheads.

[2] Cohn, Pin Tutorial, 2009

Dynamic Solution?

- **Can we perform an efficient mapping dynamically?**
- Know program behavior, cores' resources, etc.
- Dynamically instrumentation/monitoring is expensive
  - ex: applying Pin to an application has $\geq$ 50% overhead[2]
  - Instrumentation code can introduce high overheads.
- **Idea:** Move work from dynamic to static analysis (hybrid)

[2] Cohn, Pin Tutorial, 2009

Hybrid Analysis Overview

Problem: Match code to cores based on resources needed/provided

Hybrid Analysis Overview

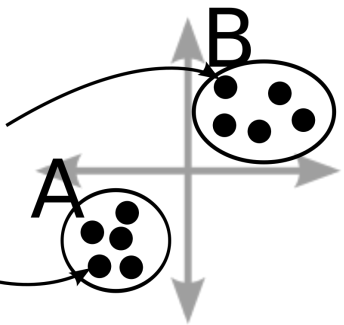
Start with program

```
void foo(){
  for(i=0:n)
  //cpu intense
  for(i=0:n)
  //mem intense
}
```

Hybrid Analysis Overview

Cluster code into groups of similar behavior

```
void foo(){
  for(i=0:n)
  //cpu intense
  for(i=0:n)
  //mem intense
}
```

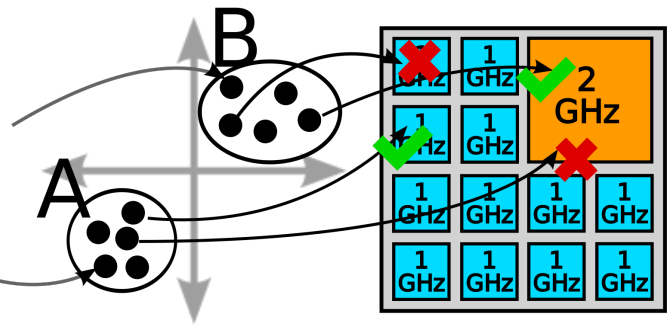


Advantage: No need to predict actual behavior, just similarity

Hybrid Analysis Overview

Run some program segments on each core type

```
void foo(){
  for(i=0:n)
  //cpu intense
  for(i=0:n)
  //mem intense
}
```

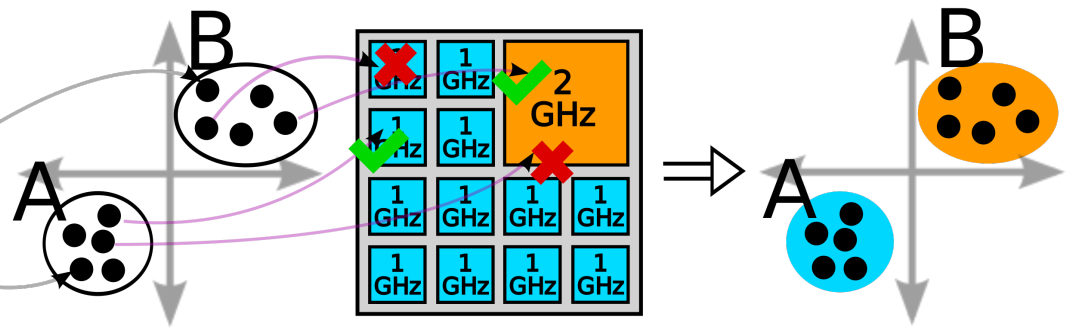


Advantage: Dynamically, no need to monitor all the code.

Hybrid Analysis Overview

Determine preferred mapping of each cluster

```
void foo(){
  for(i=0:n)
  //cpu intense
  for(i=0:n)
  //mem intense
}
```

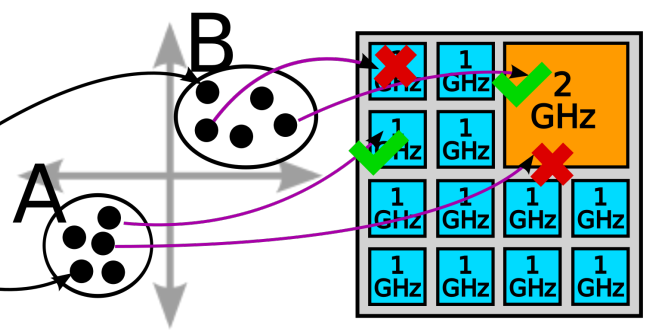


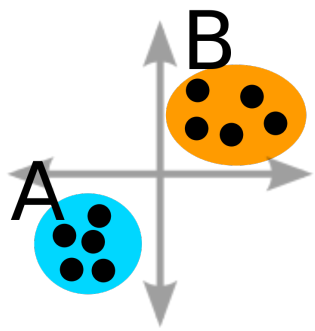
Advantage: Dynamically, no need to monitor all the code.

Hybrid Analysis Overview

Code now knows preferred core type

```
void foo(){
  for(i=0:n)
  //cpu intense
  for(i=0:n)
  //mem intense
}
```



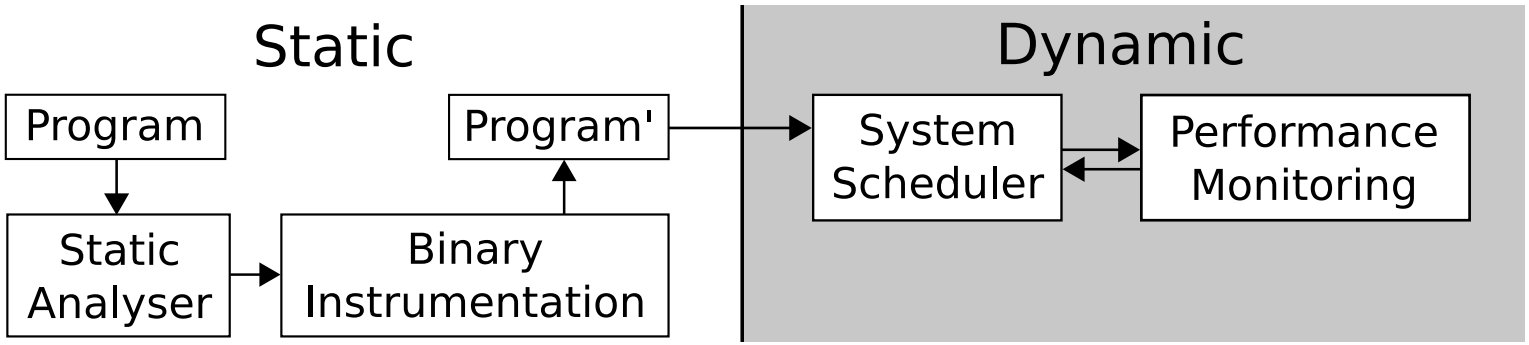



```
void foo(){
  opt(B)
  for(i=0:n)
  //cpu intense
  opt(A)
  for(i=0:n)
  //mem intense}
```

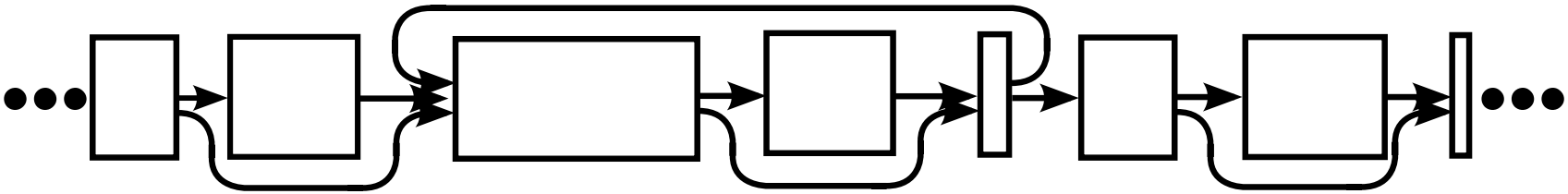
Approach Overview

Idea: Apply the same thread-to-core mapping to all approximately similar sections of code

- Statically
  - break the program into sections of code
  - determine approximate similarity between these sections
  - instrument where behavior changes
- Dynamically
  - monitor a few sections
  - make mapping decisions for similar sections

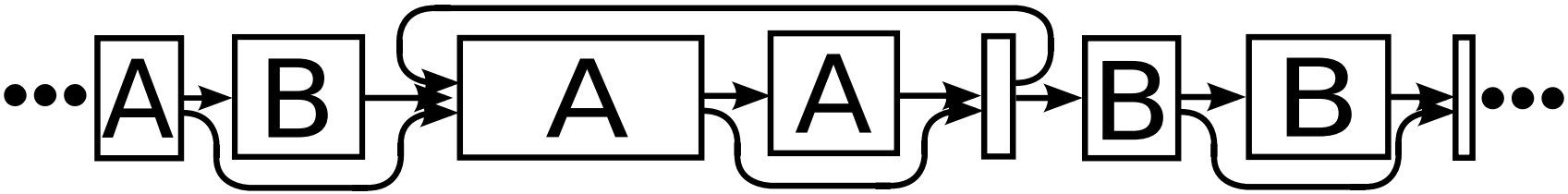


Static: Program

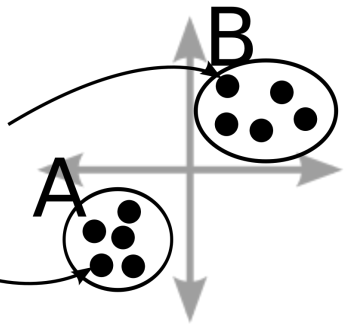


```
void foo(){
  for(i=0:n)
  //cpu intense
  for(i=0:n)
  //mem intense
}
```

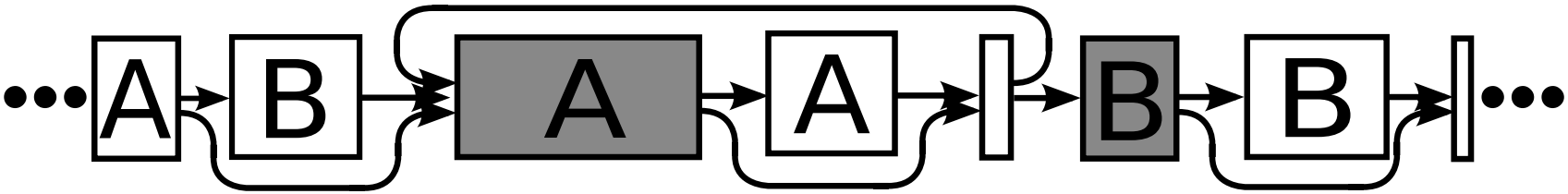
Static: Determine approximate similarity



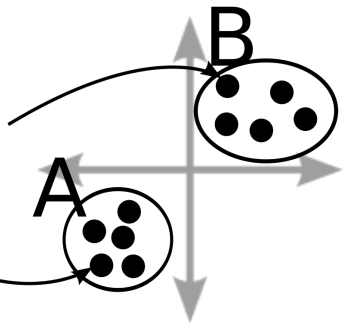

```
void foo(){
  for(i=0:n)
  //cpu intense
  for(i=0:n)
  //mem intense
}
```



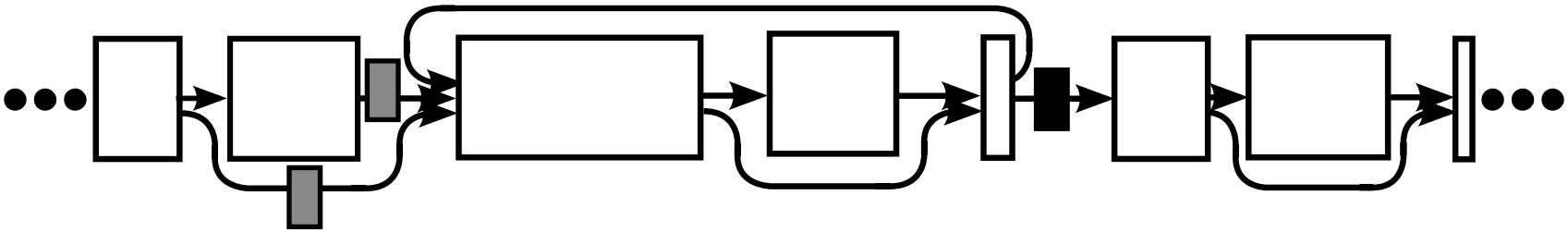
Static: Reduce number of transition points



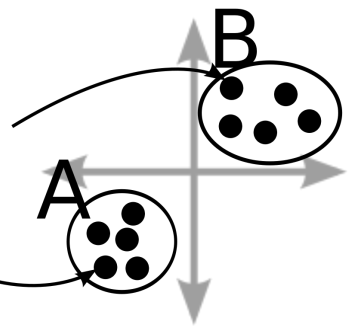

```
void foo(){
  for(i=0:n)
  //cpu intense
  for(i=0:n)
  //mem intense
}
```



Static: Insert phase marks



```
void foo(){
  for(i=0:n)
  //cpu intense
  for(i=0:n)
  //mem intense
}
```

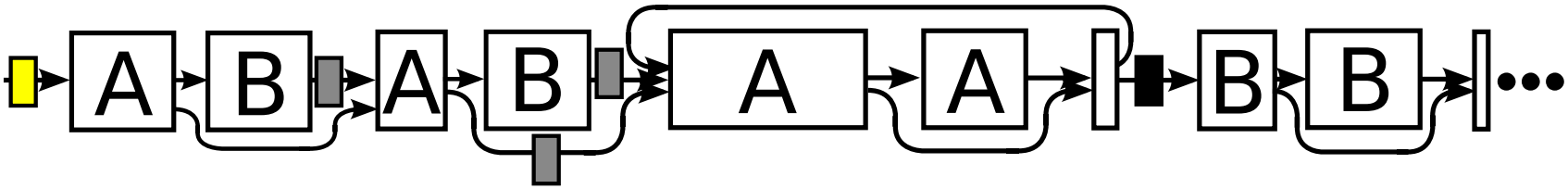



```
void foo(){
  opt(B)
  for(i=0:n)
  //cpu intense
  opt(A)
  for(i=0:n)
  //mem intense}
```

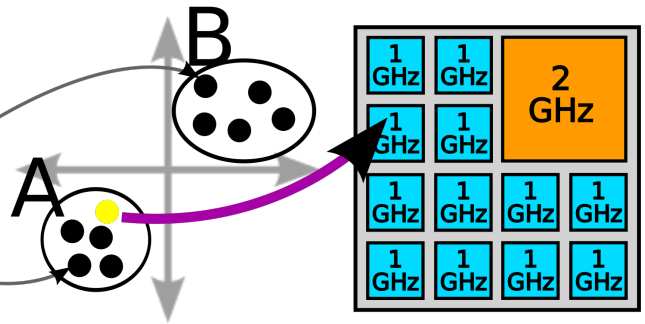
Static/Dynamic: Phase Marks

- Inserted on path where behavior is likely to change
  - Must pick “good” points to avoid overhead
  - For example, loop entry points
- Contains type information
- Contains dynamic analysis code
  - Monitor behavior if mapping is unknown
  - Switch cores if mapping is known

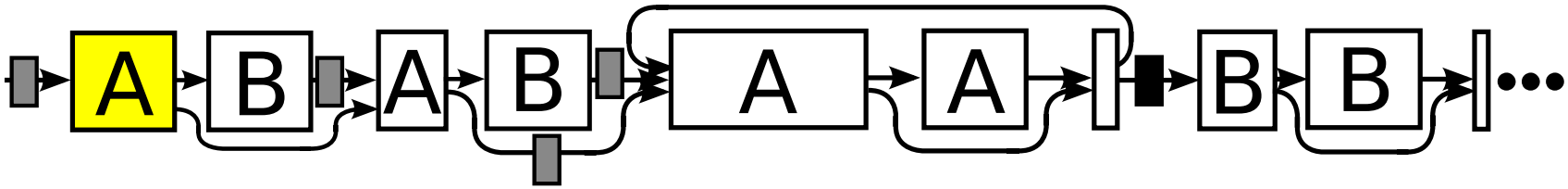
Dynamic: Monitor



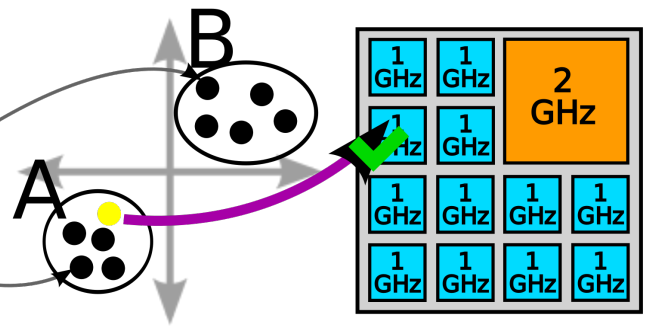

```
void foo(){
  for(i=0:n)
  //cpu intense
  for(i=0:n)
  //mem intense
}
```



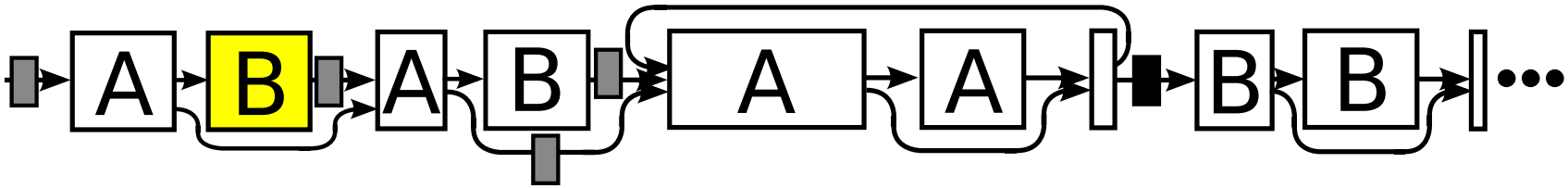
Dynamic: Run



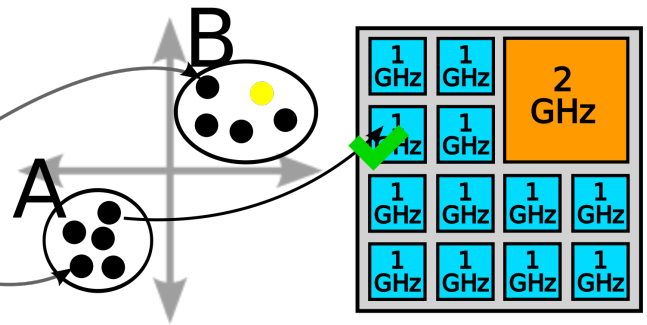

```
void foo(){
  for(i=0:n)
  //cpu intense
  for(i=0:n)
  //mem intense
}
```



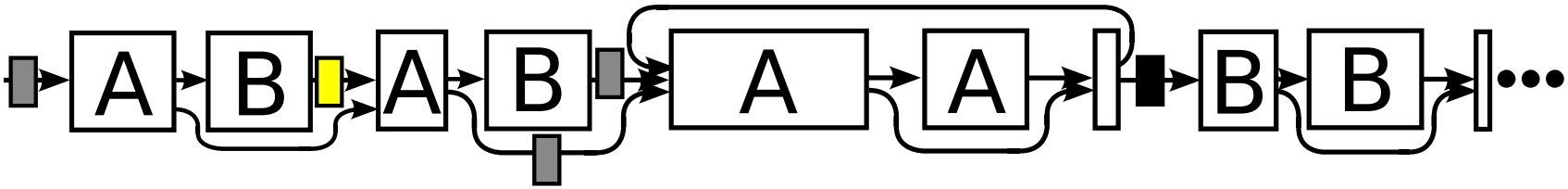
Dynamic: Run



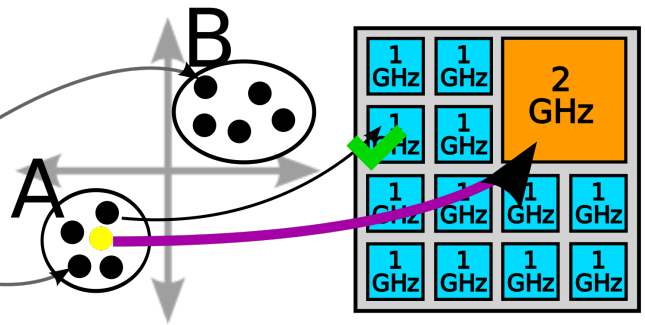

```
void foo(){
  for(i=0:n)
  //cpu intense
  for(i=0:n)
  //mem intense
}
```



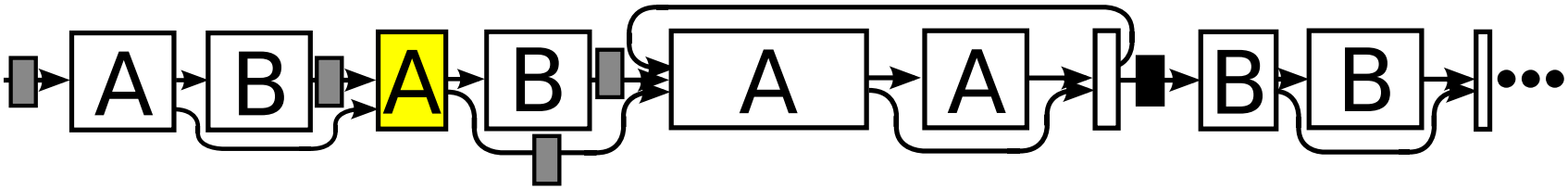
Dynamic: Monitor



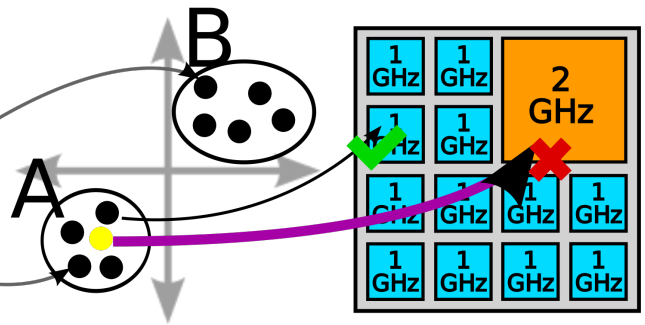

```
void foo(){
  for(i=0:n)
  //cpu intense
  for(i=0:n)
  //mem intense
}
```



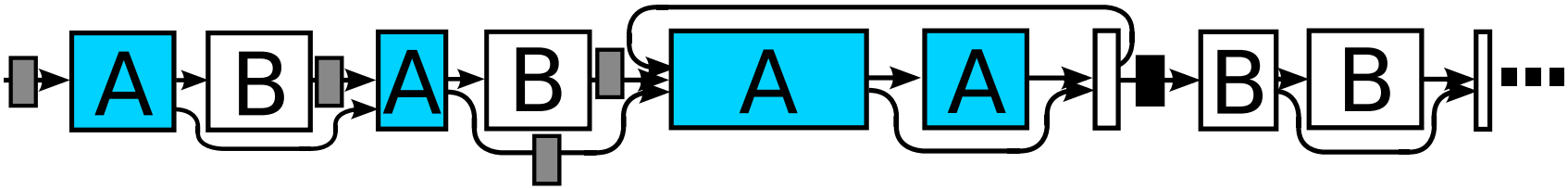
Dynamic: Run



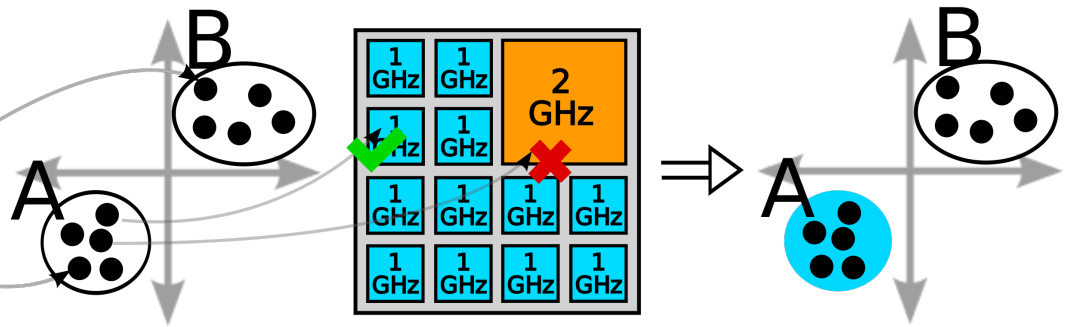

```
void foo(){
  for(i=0:n)
  //cpu intense
  for(i=0:n)
  //mem intense
}
```



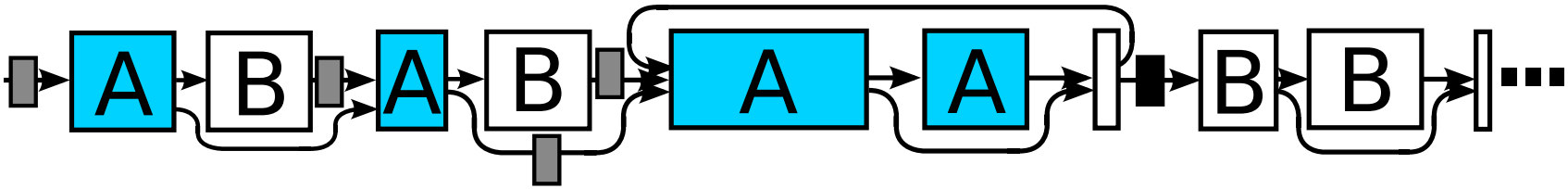
Dynamic: Determine preferred core



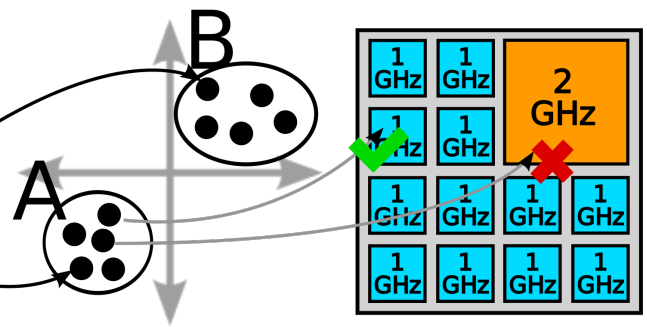

```
void foo(){
  for(i=0:n)
  //cpu intense
  for(i=0:n)
  //mem intense
}
```

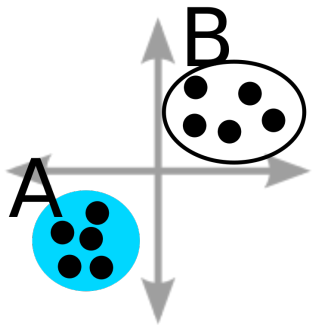


Dynamic: Determine preferred core



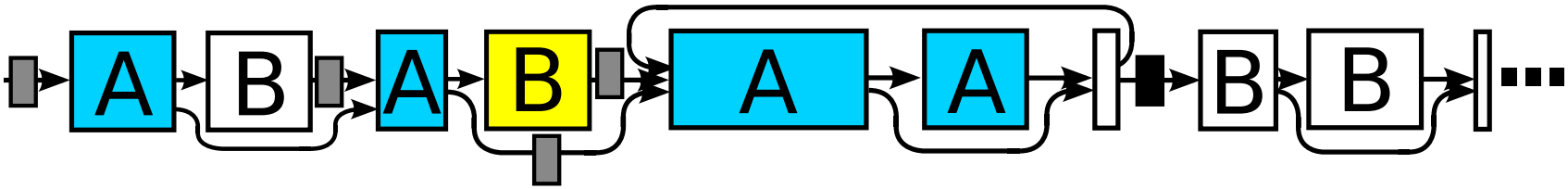

```
void foo(){
  for(i=0:n)
  //cpu intense
  for(i=0:n)
  //mem intense
}
```



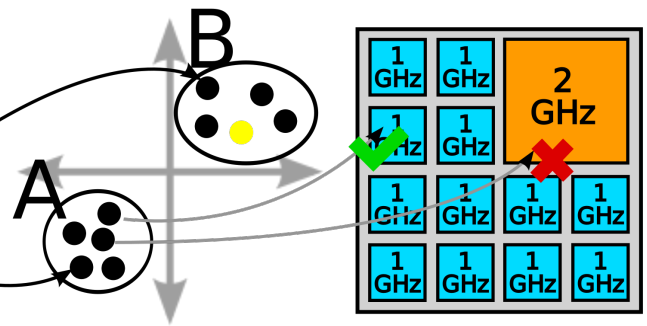


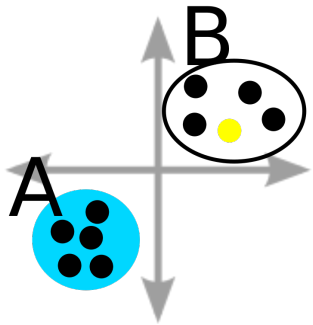

```
void foo(){
  opt(B)
  for(i=0:n)
  //cpu intense
  opt(A)
  for(i=0:n)
  //mem intense}
```

Dynamic: Run



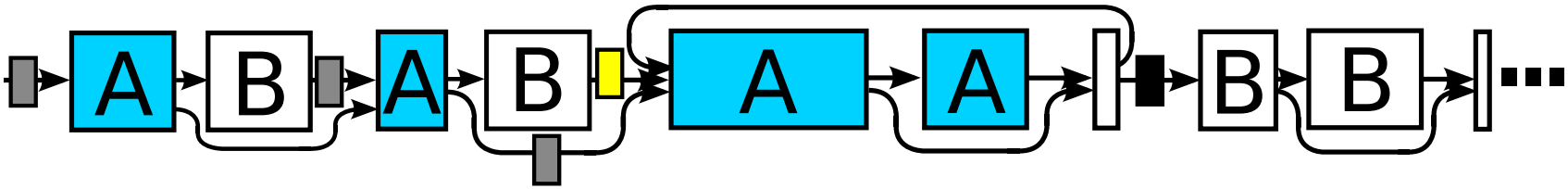

```
void foo(){
  for(i=0:n)
  //cpu intense
  for(i=0:n)
  //mem intense
}
```



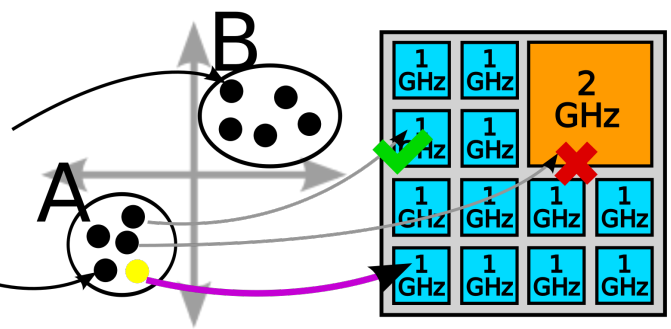


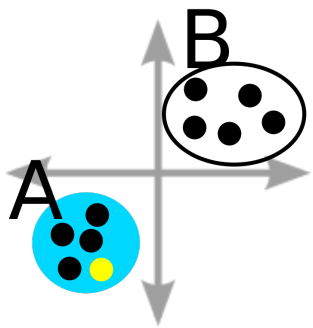

```
void foo(){
  opt(B)
  for(i=0:n)
  //cpu intense
  opt(A)
  for(i=0:n)
  //mem intense}
```

Dynamic: Switch to matched core (slow)



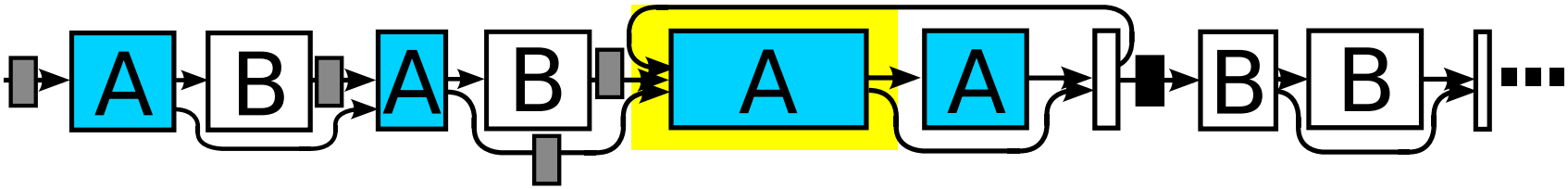

```
void foo(){
  for(i=0:n)
  //cpu intense
  for(i=0:n)
  //mem intense
}
```



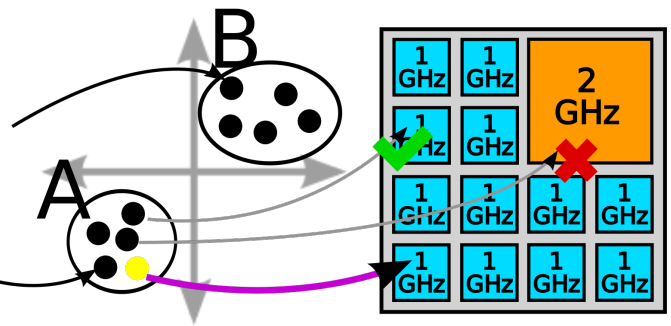


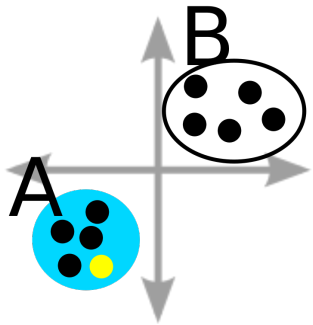

```
void foo(){
  opt(B)
  for(i=0:n)
  //cpu intense
  opt(A)
  for(i=0:n)
  //mem intense}
```

Dynamic: Run on matched core (slow)




```
void foo(){
  for(i=0:n)
  //cpu intense
  for(i=0:n)
  //mem intense
}
```



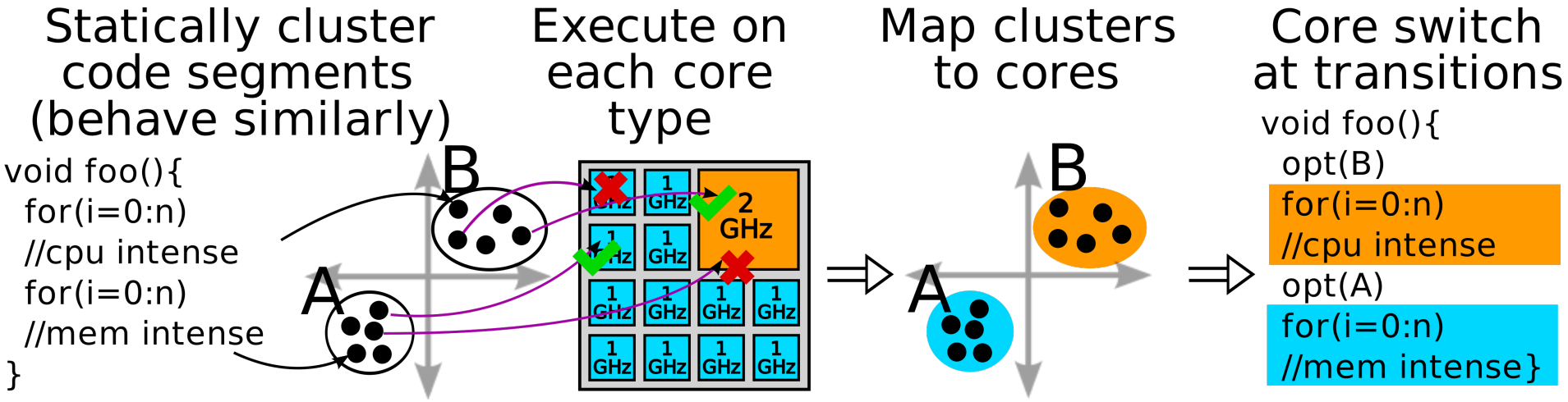



```
void foo(){
  opt(B)
  for(i=0:n)
  //cpu intense
  opt(A)
  for(i=0:n)
  //mem intense}
```

Solution overview

Hybrid analysis – phase-guided tuning

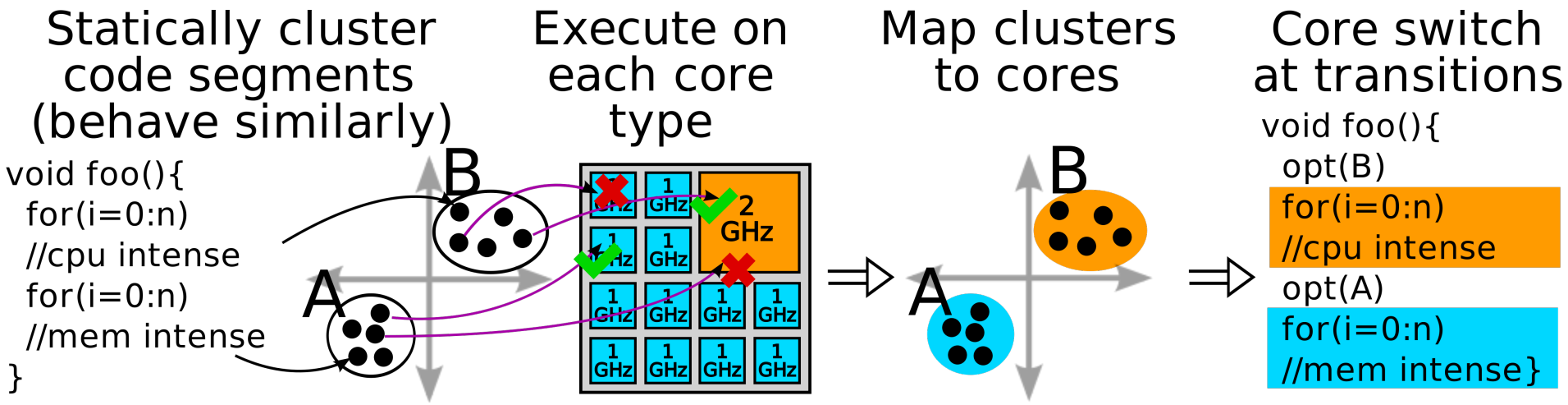
- Statically predict approximate similarity



Solution overview

Hybrid analysis – phase-guided tuning

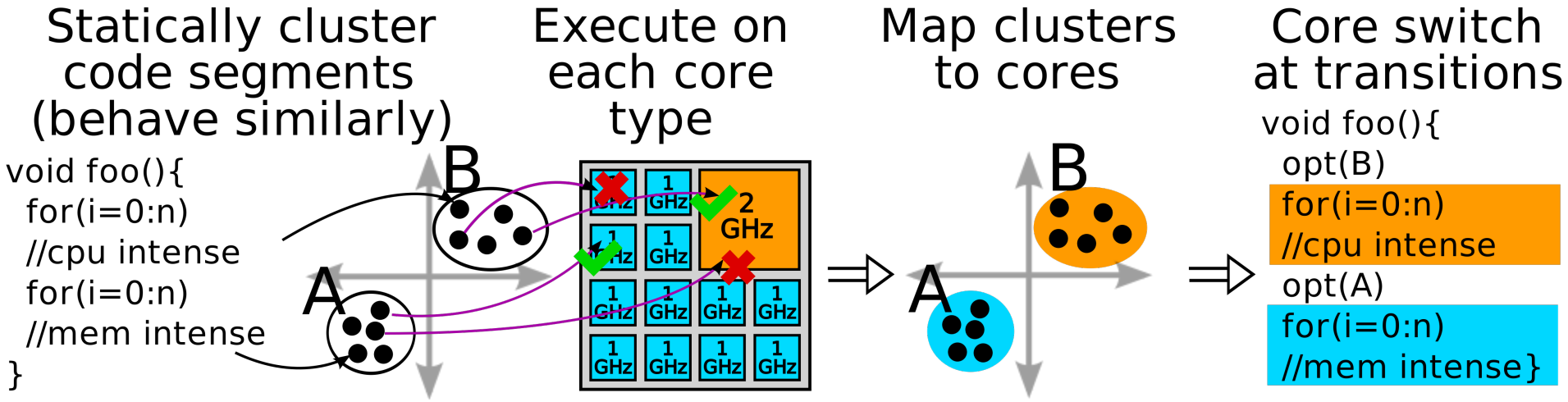
- Statically predict approximate similarity
- Statically instrument application with this information
  - Find likely phase transitions (structures with different types)
  - Instrument paths into likely long running code (e.g. loops)



Solution overview

Hybrid analysis – phase-guided tuning

- Statically predict approximate similarity
- Statically instrument application with this information
  - Find likely phase transitions (structures with different types)
  - Instrument paths into likely long running code (e.g. loops)
- Behavior of a section gives insight into entire cluster



Experimental Setup

- Hardware setup: Quad Core - 2x2.4GHz, 2x1.6GHz
- Workloads
  - 18-84 SPEC CPU2000 and CPU2006 benchmarks
  - constant workload size
- Compare to standard Linux assignment

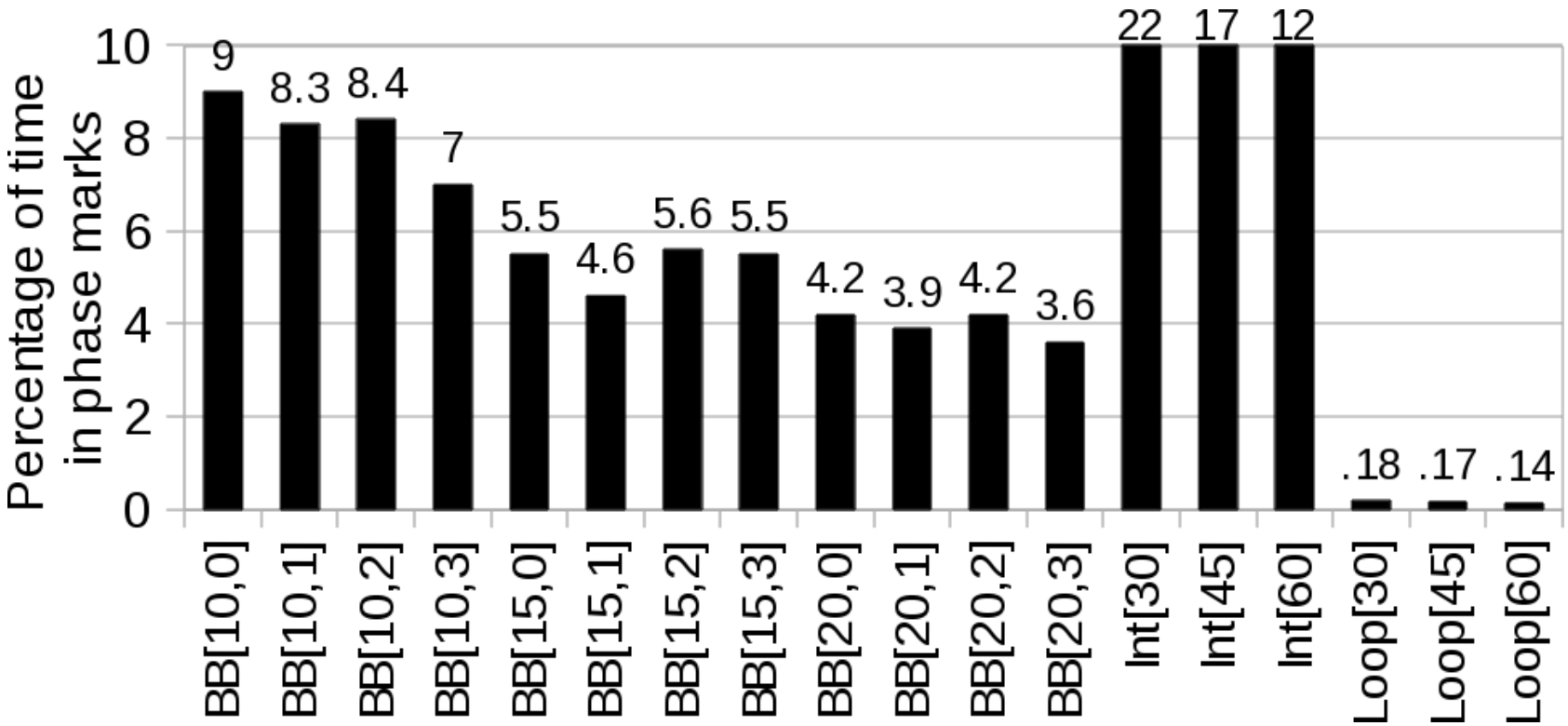
Overhead

- Space overhead
- Time overhead
- Average cycles per switch

Time Overhead

- Time overhead
  - Time spent executing code in phase marks
  - Directly impacts performance

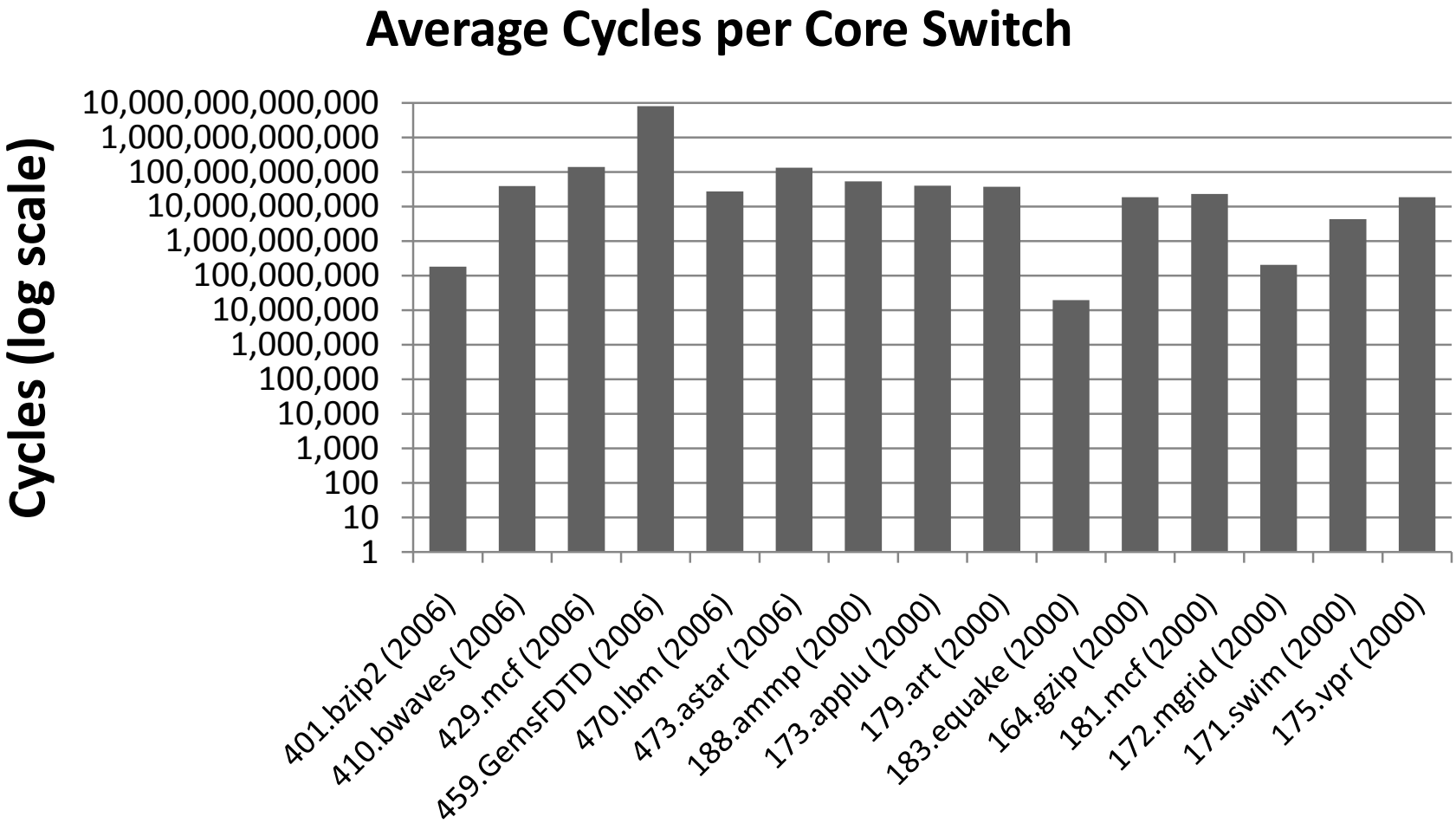
Time Overhead



Cycles per Switch

- Average cycles per switch
  - Impacts performance
  - Must be high enough to amortize runtime costs

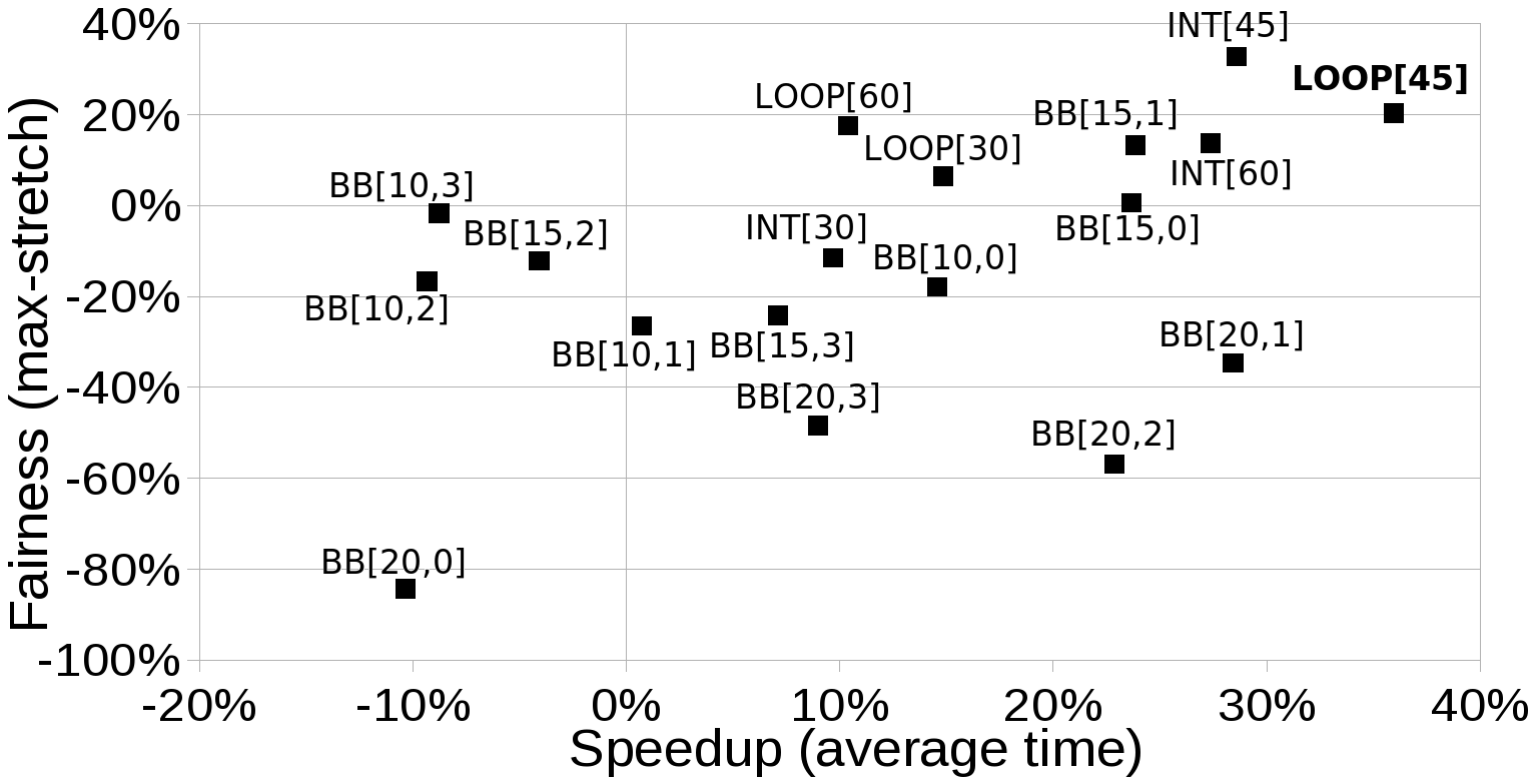
Cycles per Switch



Performance

- Speedup – average process time
- Fairness (max-stretch) – maximum process slow-down

Speedup vs Fairness



Best: Inter-procedural loop technique, min. size 45 instructions

Future Work

- Improve static behavior similarity prediction

Future Work

- Improve static behavior similarity prediction
- Improved techniques for picking phase marks

  e.g. estimate number of iterations instead of number of instructions

Future Work

- Improve static behavior similarity prediction
- Improved techniques for picking phase marks

  e.g. estimate number of iterations instead of number of instructions
- Dynamic optimization
  - feedback mechanism to improve assignment
  - globally optimal assignment

Conclusion

- Performance asymmetric multicores are beneficial
- Problem: Techniques to effectively utilize are needed
- **Idea: Use repeating behavior to reduce dynamic overhead.**

Conclusion

- Performance asymmetric multicores are beneficial
- Problem: Techniques to effectively utilize are needed
- **Idea: Use repeating behavior to reduce dynamic overhead.**
  - Programmer oblivious – behavior and architectures
  - Automatic – requires no assistance from programmer
  - Negligible overhead – less than 0.2% runtime overhead
  - Transparent deployment – no OS or compiler modification
  - Tune once run anywhere – architecture independent

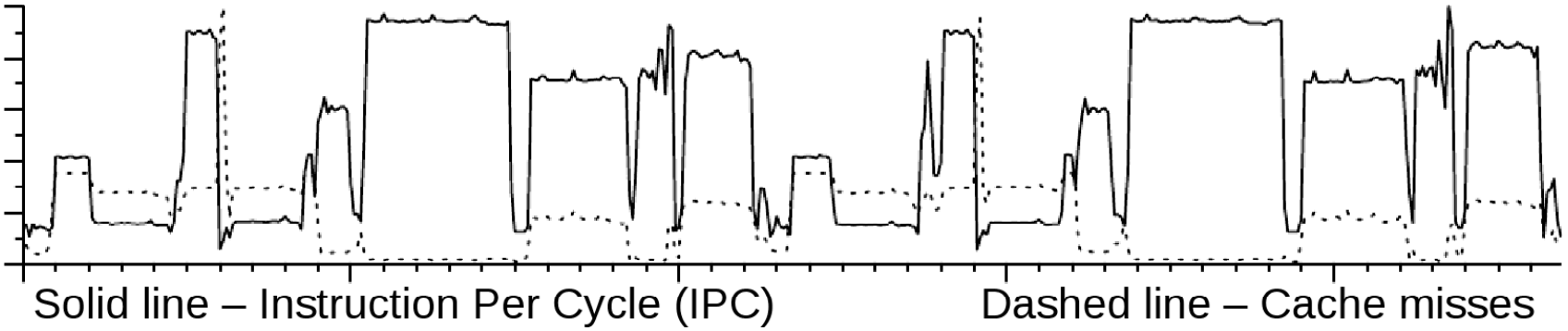
Questions

Questions?

Idea

Problem: Match code to cores based on resources needed/provided

- **Behavior:** resource requirements (IPC, cache miss, etc.)
- **Phase:** segment of execution with similar behavior throughout[3]
- **Insight**: Behavior tends to repeat itself.



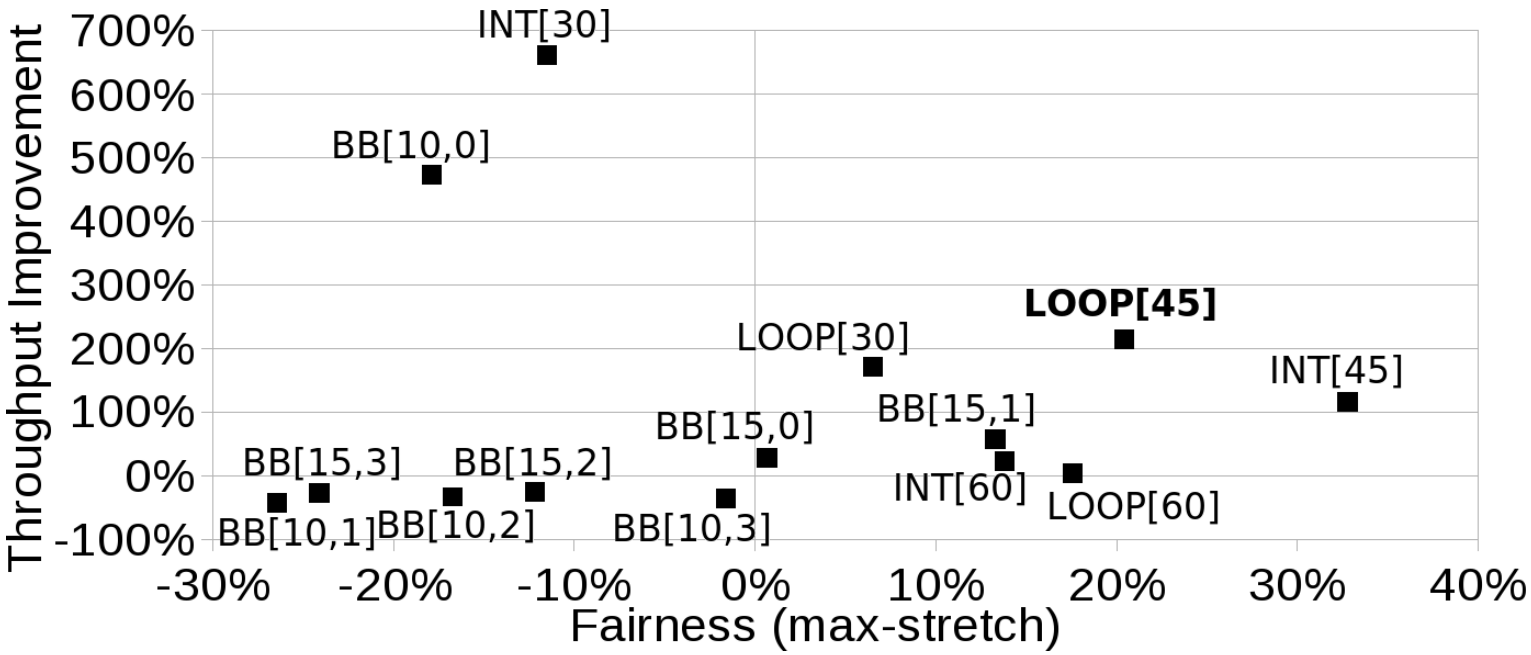
Phase behavior for gcc (taken from [3]) 5

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

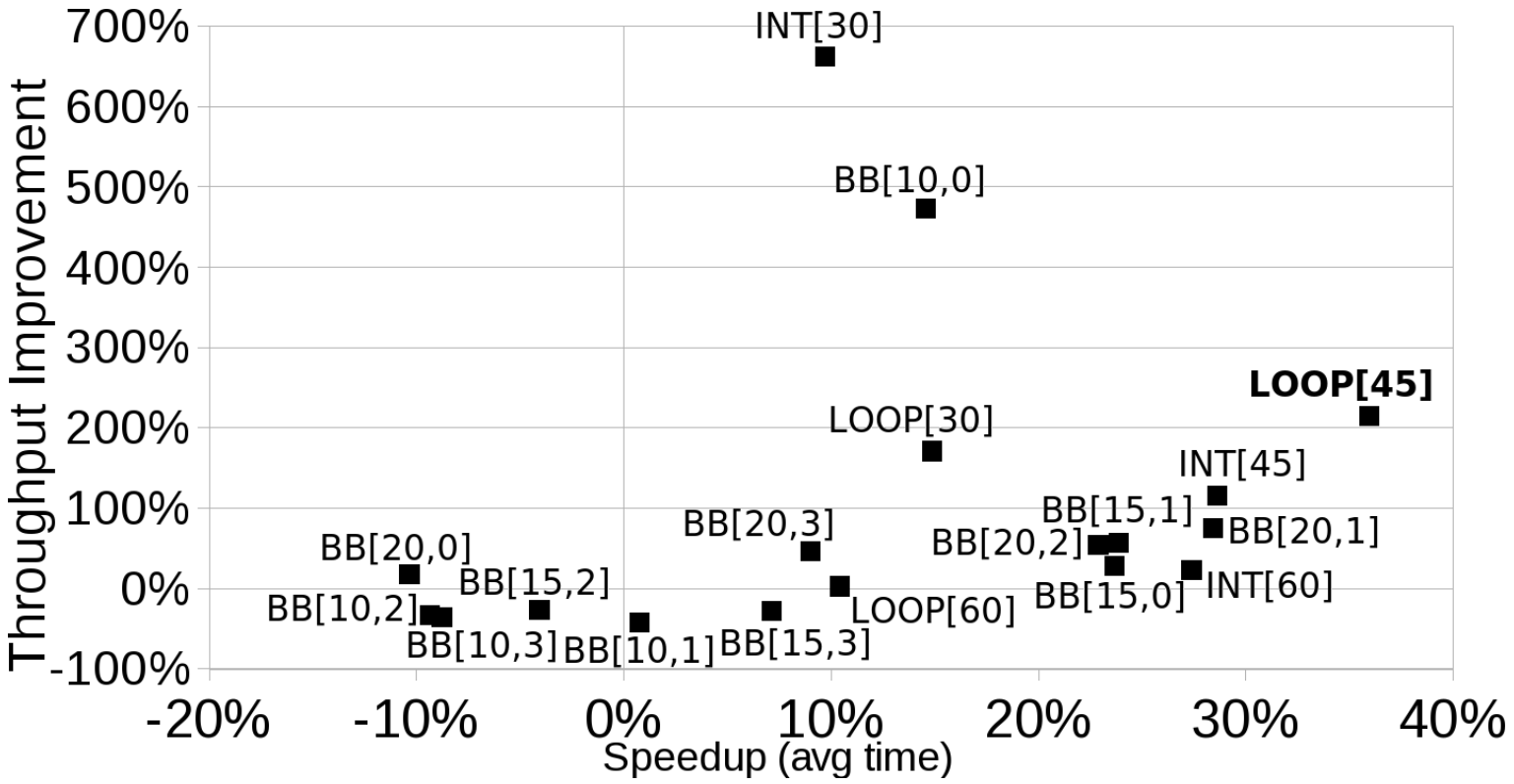
Experimental Setup

- 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
    - 18-84 SPEC CPU2000 and CPU2006 benchmarks
    - constant workload size
- Compare to standard Linux assignment

Throughput vs Fairness



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

Phase Marking cost

Recall that we had two problems to solve:

- Move work from dynamic to static analysis – phase-guided
- Reduce number of / pick good instrumentation points

Phase Marking cost

Recall that we had two problems to solve:

- Move work from dynamic to static analysis – phase-guided
- Reduce number of / pick good instrumentation points

Phase Marking cost

- Phase marks cost (space and run-time)
- We need techniques to pick good insertion points
  - Basic block
  - Basic block with look-ahead
  - Interval (intra-procedural)
  - Loop (inter-procedural)

5

Basic Block

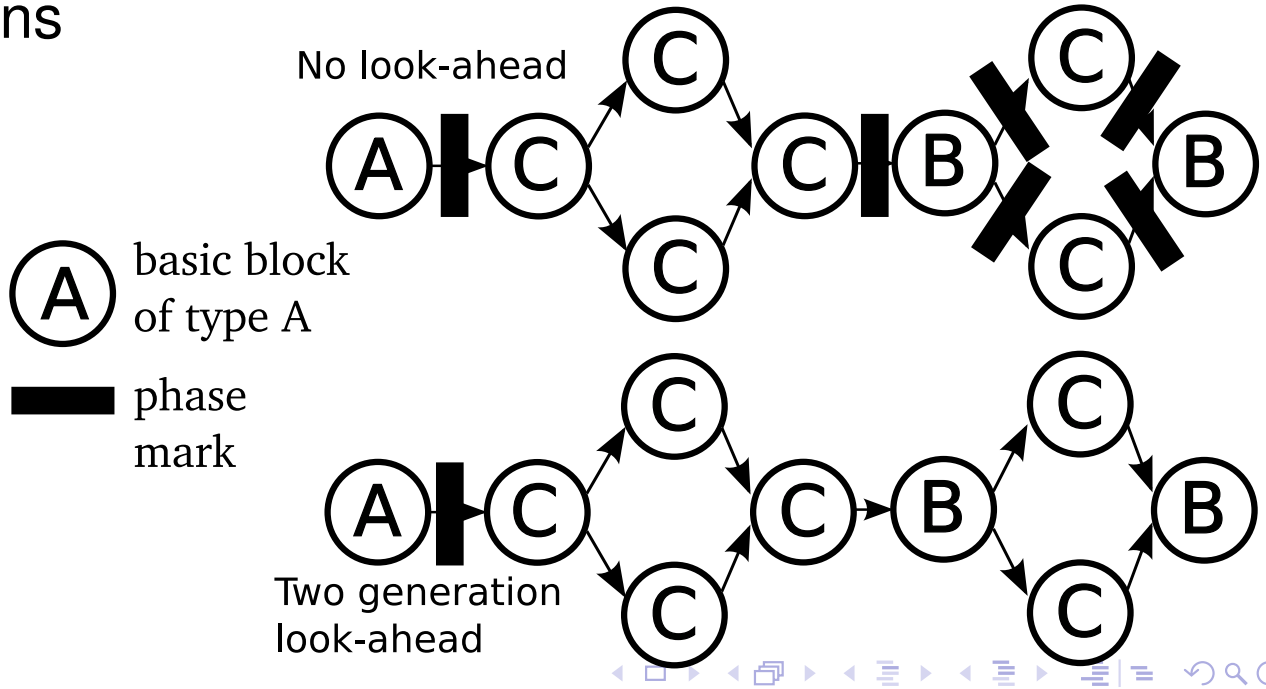
Basic block

- Similarity is done on basic blocks
- Transitions between blocks with different types
- Problem: blocks are small, cost is likely higher than gains

Basic Block with Look-ahead

Basic block with look-ahead

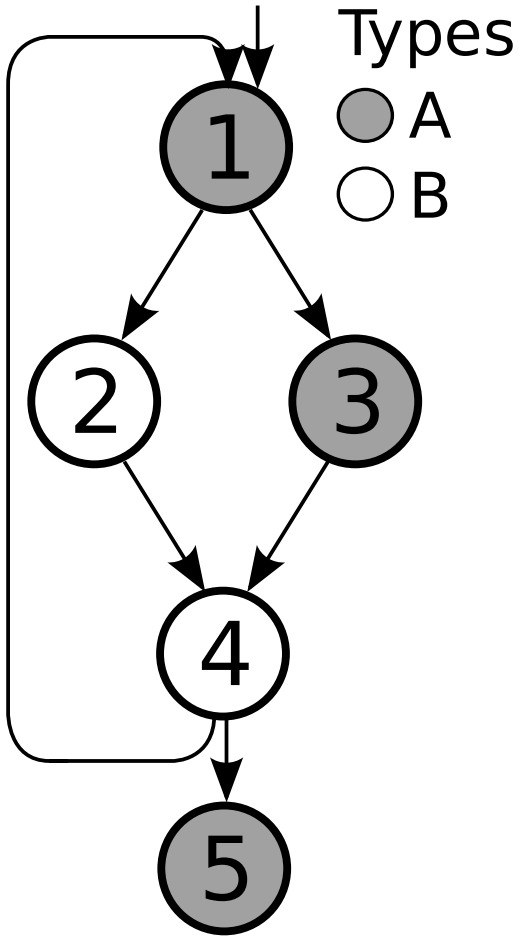
- Instrument if next ***n*** blocks are of similar type
- Ensures switching for larger number of instructions
- In some cases, captures loops
- Problem: allows multiple switches in small loops
- Problem: cost of optimizing a few blocks is still likely to overshadow gains



Intervals

Interval (intra-procedural)

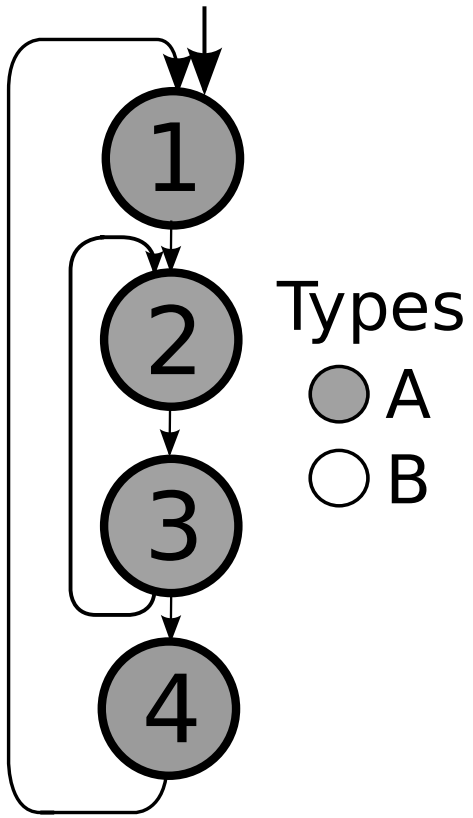
- Instrument if interval has a predominant type
- Intervals capture small loops
- All instructions are part of some interval
- Problem: some intervals are not loops (few instructions)



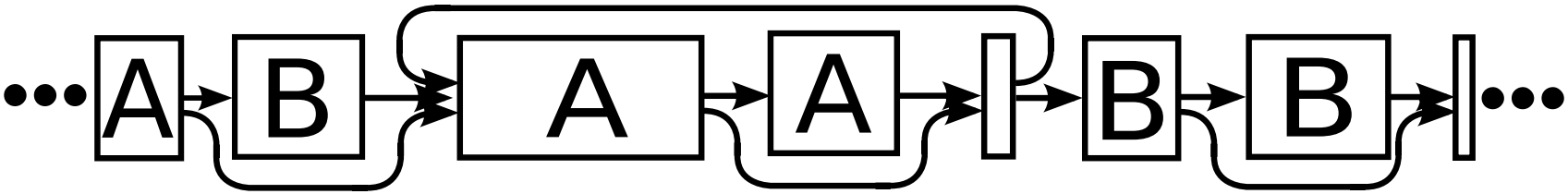
Loop

Loop based (inter-procedural)

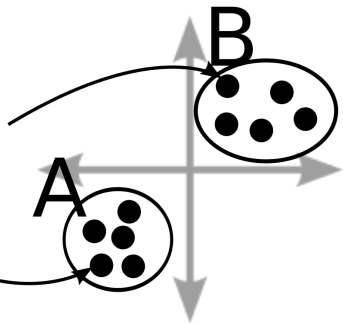
- Instrument loop based on predominant type
- Remove instrumentation if part of larger loop of same type
- Captures loops and nothing else
- Considers nested loops
- Considers function calls



Static: Determine approximate similarity




```
void foo(){
  for(i=0:n)
  //cpu intense
  for(i=0:n)
  //mem intense
}
```



Static: Predicting Similarity

- Not predicting actual behavior, just similarity
- Various metrics to consider
  - **instruction mix** (int vs float, div, etc)
  - **cache behavior**
  - branch prediction accuracy
  - available ILP
  - data structure(s)
- Proof-of-concept: 85% accuracy using few metrics

Space overhead

- Space overhead
  - May hurt instruction cache performance
  - Increased binary file size

Space Overhead