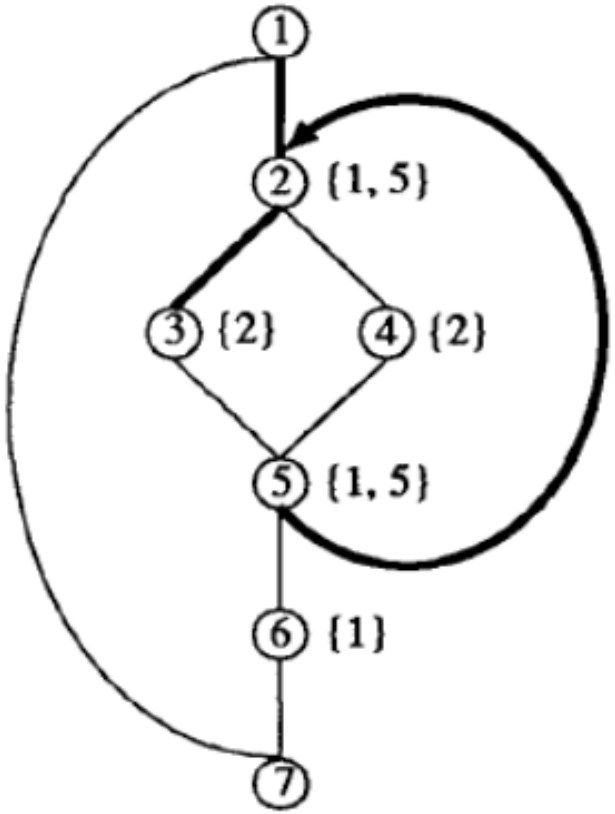
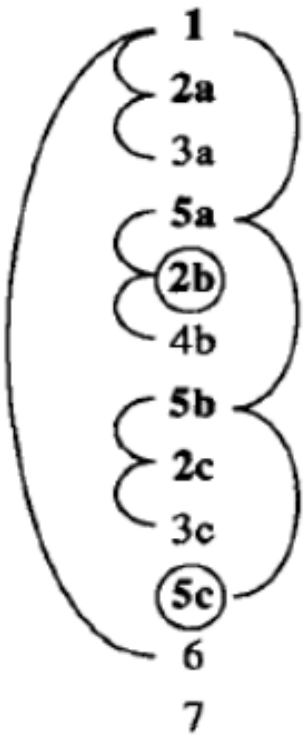
Program Analysis





Jose Lugo-Martinez

CSE 240C: Advanced Microarchitecture

Prof. Steven Swanson

Outline

- **Motivation**
  - ILP and its limitations
  - Previous Work
- **Limits of Control Flow on Parallelism**
  - Author's Goal
  - Abstract Machine Models
  - Results & Conclusions
- **Automatically Characterizing Large Scale Program Behavior**
  - Author's Goal
  - Finding Phases
  - Results & conclusions

What is ILP?

Instructions that do not have dependencies on each other; can be executed in any order

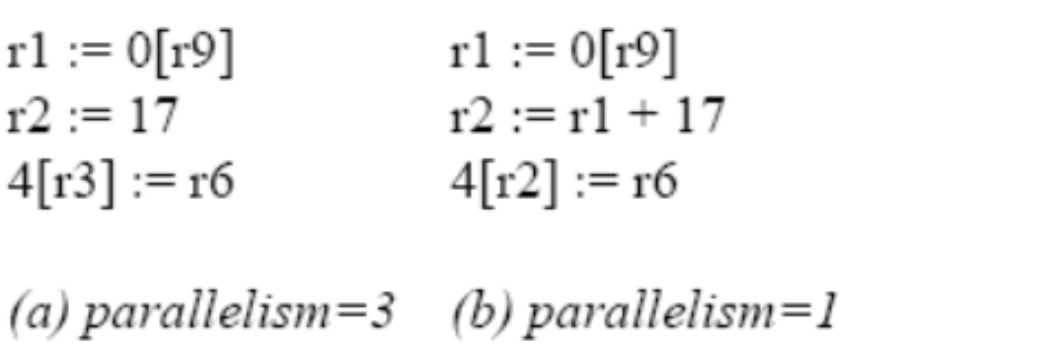


Figure 1: Instruction-level parallelism (and lack thereof)

How much parallelism is there?

That depends how hard you want to look for it...

Ways to increase ILP:

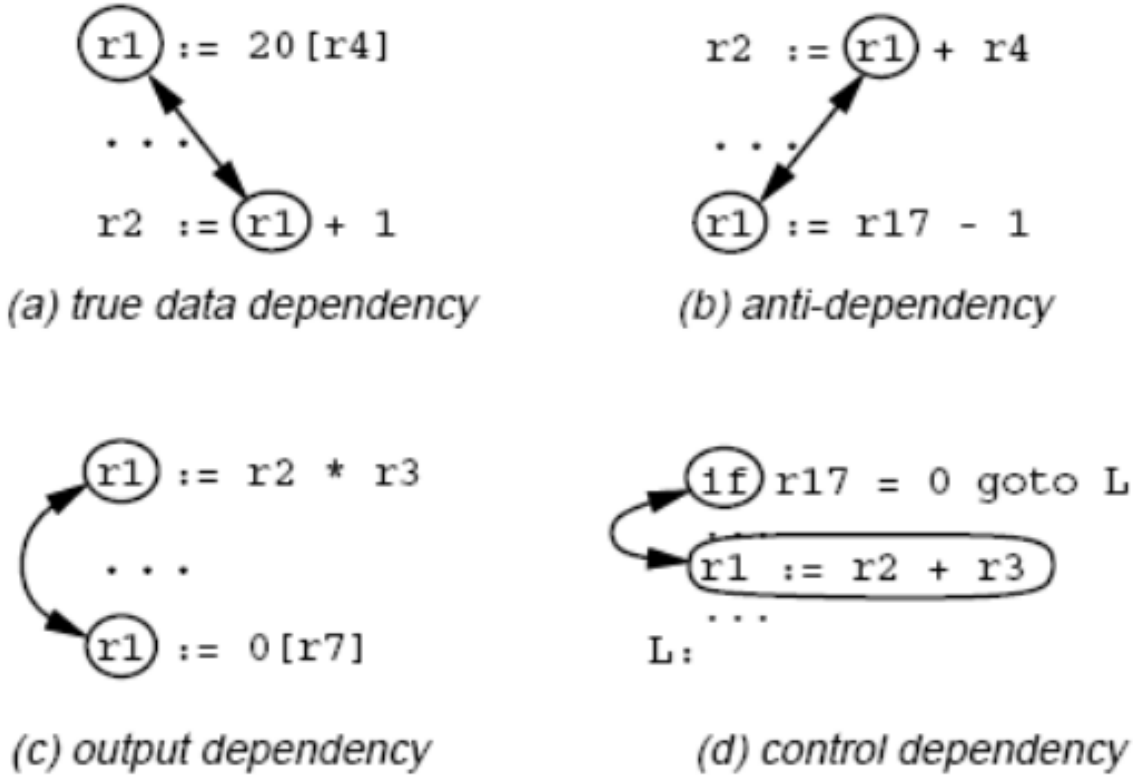
- Register renaming
- Alias analysis
- Branch prediction
- Loop unrolling

ILP Challenges and Limitations to Multi-Issue Machines

- In order to achieve parallelism we should not have dependencies among instructions which are executing in parallel

- Inherent limitations of ILP
  - 1 branch every 5 instructions
  - Latencies of units: many operations must be scheduled
  - Increase ports to Register File (bandwidth)
  - Increase ports to memory (bandwidth)
  - …

Dependencies



Previous Work: Wall's Study Overall results

	branch predict	*jump predict*	*reg renaming*	*alias analysis*
Stupid	none	none	none	none
Fair	infinite	infinite	256	inspection
Good	infinite	infinite	256	perfect
Great	infinite	infinite	perfect	perfect
Perfect	perfect	perfect	perfect	perfect

Figure 4. Five increasingly ambitious models.

parallelism = # of instr / # cycles required

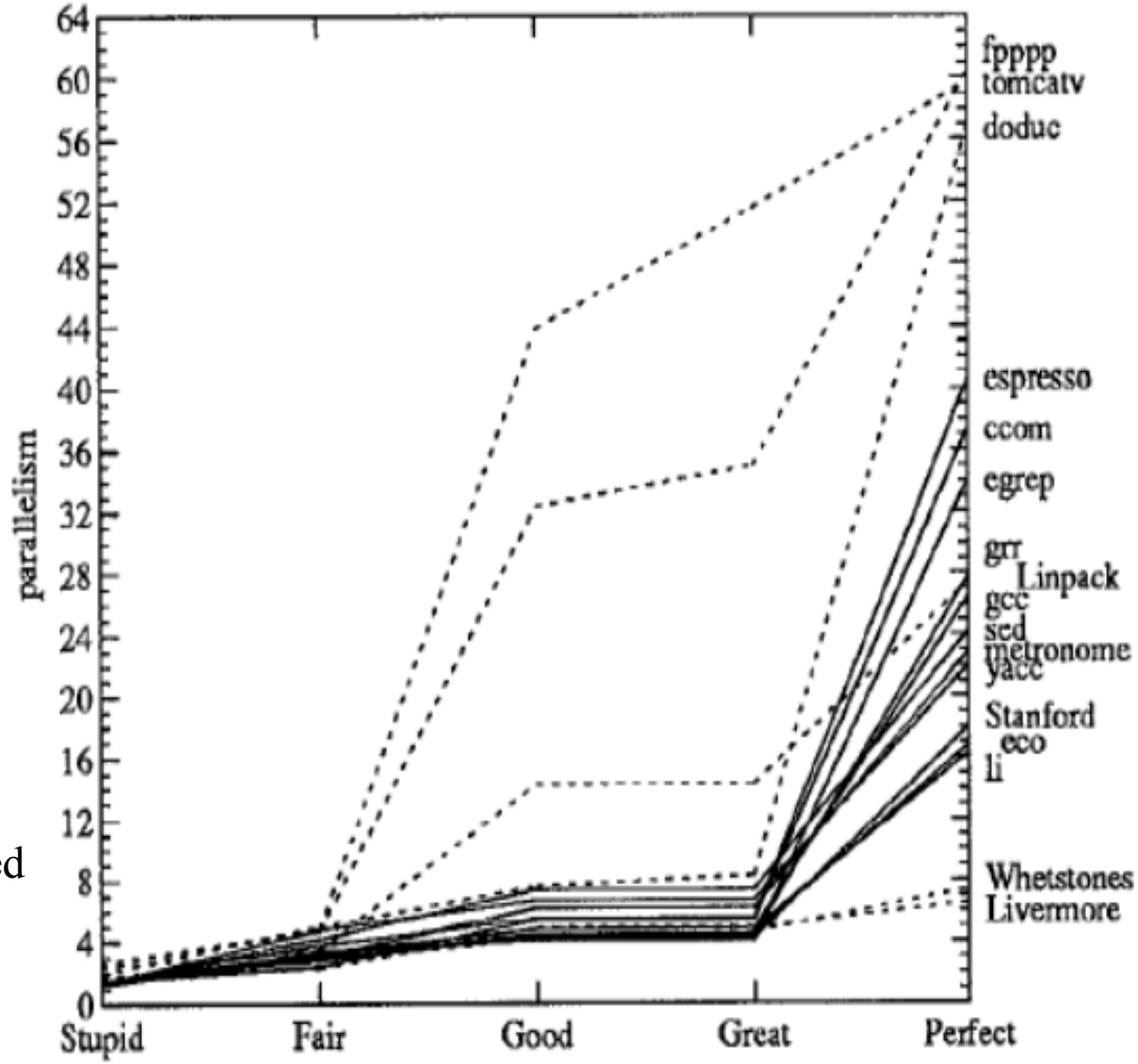
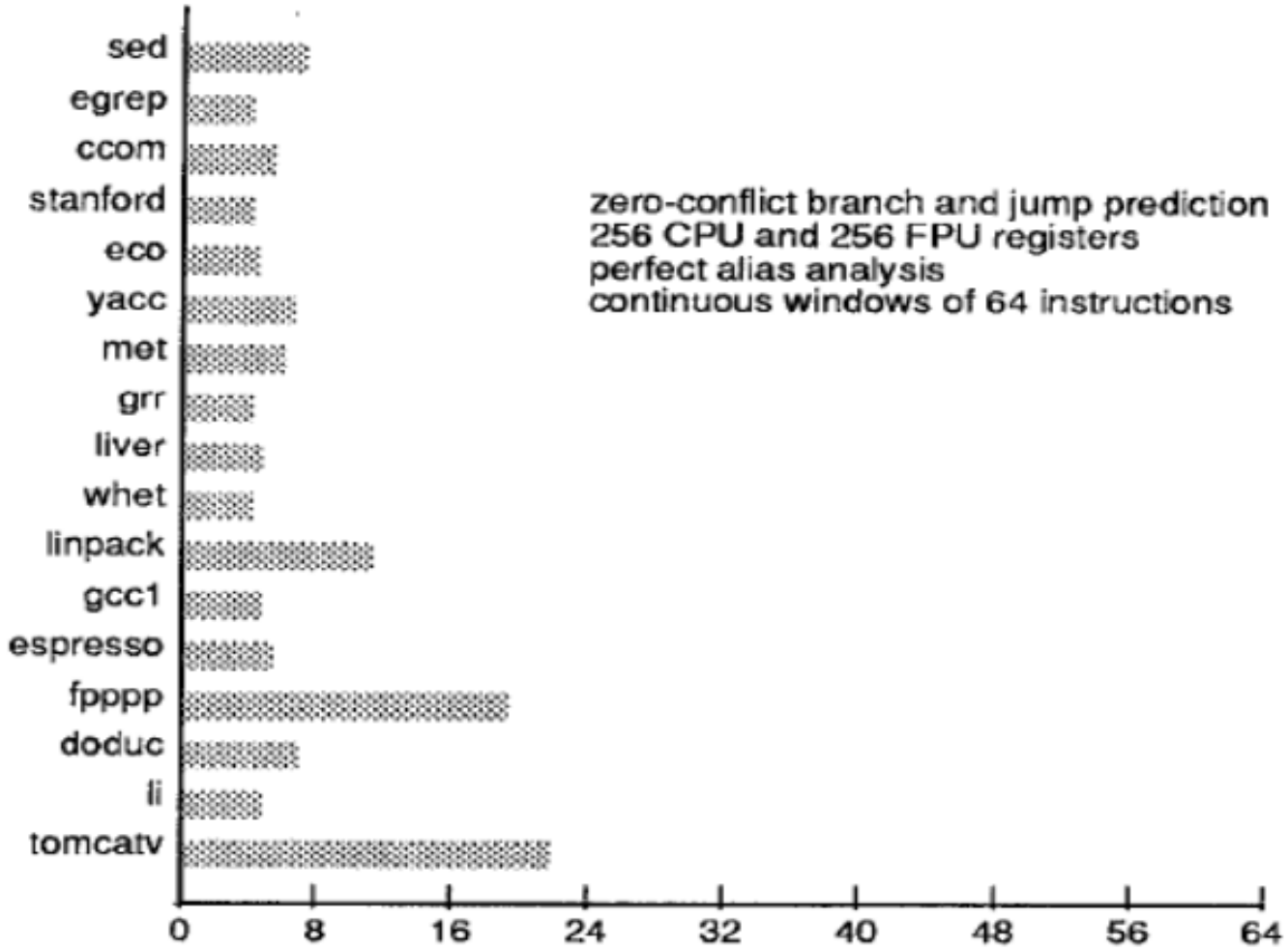


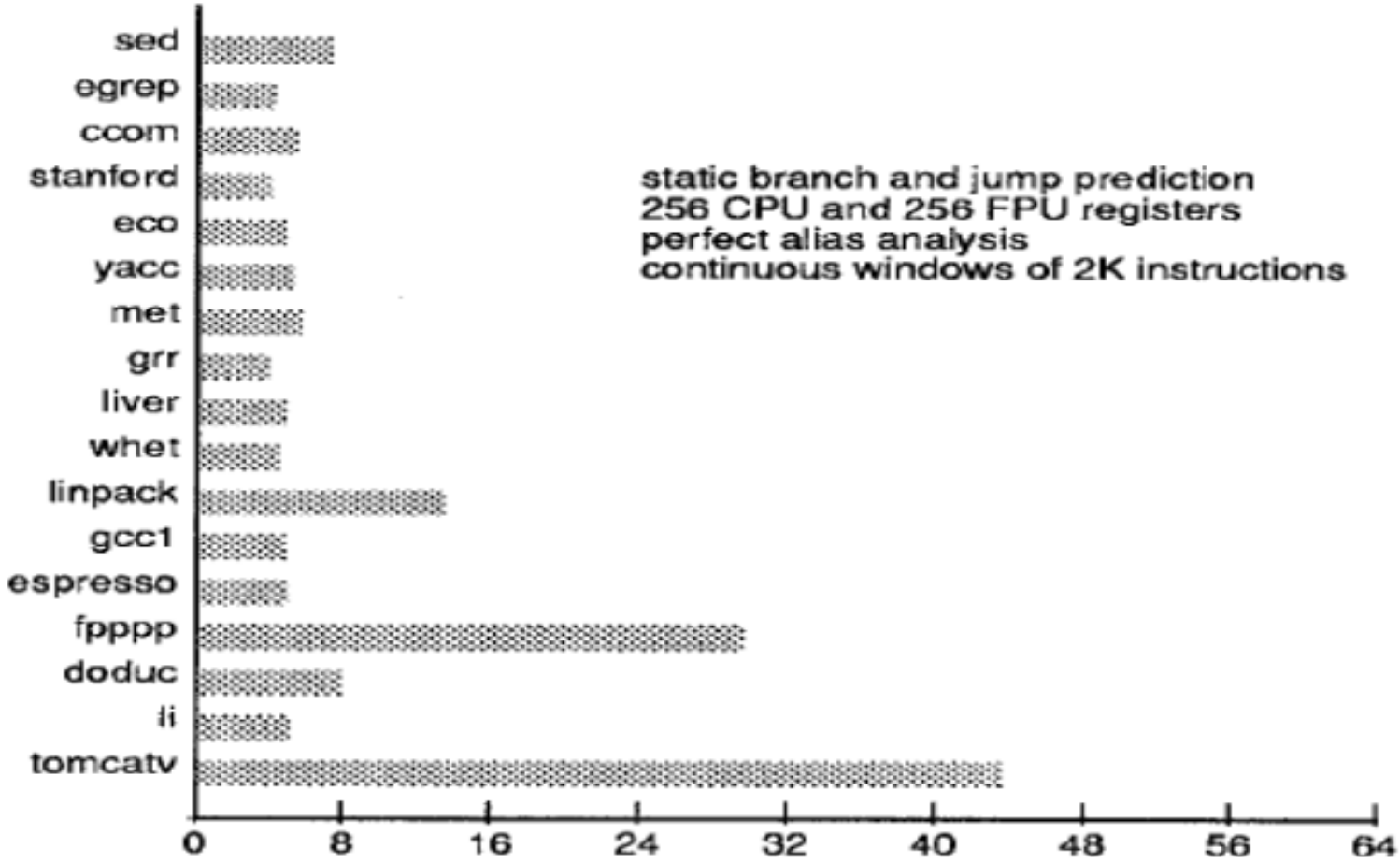
Figure 6. Parallelism under the five models.

Parallelism achieved by a quite ambitious HW style model



The average parallelism is around 7, the median around 5.

Parallelism achieved by a quite ambitious SW style model



The average parallelism is close to 9, but the median is still around 5.

Wall's Study Conclusions

- The results contrasts sharply with previous experiments that assume perfect branch prediction, (i.e. *oracle*)
- His results suggests a severe limitation in current approaches
- The reported speedups for this processor on a set of non-numeric programs ranges from 4.1 to 7.4 for an aggressive HW algorithm to predict outcomes of branches

Author's Motivation and Goal

- **Motivation**: Much more parallelism is available on an oracle machine, suggesting that the bottleneck in Wall's experiment is due to control flow

- **Goal**: Discover ways to increase parallelism by an order of magnitude beyond current approaches

- How?
  - They analyze and evaluate the techniques of speculative execution, control dependence analysis, and following multiple flows of control

- Expectations?
  - Hopefully establish the inadequacy of current approaches in handling control flow and identify promising new directions

Speculation with Branch Prediction

- Speculation is a form of guessing
- Important for branch prediction
  - Need to “take our best shot” at predicting branch direction. A common technique to improve the efficiency of speculation is to only speculate on instructions from the most likely execution path
- If we speculate and are wrong, need to back up and restart execution to point at which we predicted incorrectly
- The success of this approach depends on the accuracy of branch prediction

Control Dependence Analysis

```
if (a < 0)
    b = 1;
c = 2;
```

- b = 1 is *control dependent* on the condition a < 0
- c = 2 is *control independent*
- branch on which an instruction is control dependent is the *control dependence branch*
- Control dependence in programs can be computed in a compiler using the *reverse dominance frontier* algorithm

Multiple Control Flows

```
for (i = 0; i < 100; i++)
    if (A[i] > 0) foo();
bar();
```

- In the example above, control dependence analysis shows that the *bar* function can run concurrently with the preceding loop
- However, a uniprocessor can typically only follow one flow of control at any time
- Support multiple flows of control is necessary to fully exploit the parallelism uncovered by control dependence analysis
- Multiprocessor architectures are a general means of providing this support
- Each processor can follow an independent flow of control

Abstract Machine Models

BASE An instruction cannot execute until the immediately preceding branch in the trace is resolved. This implies that branch instructions must execute in order, one per cycle.

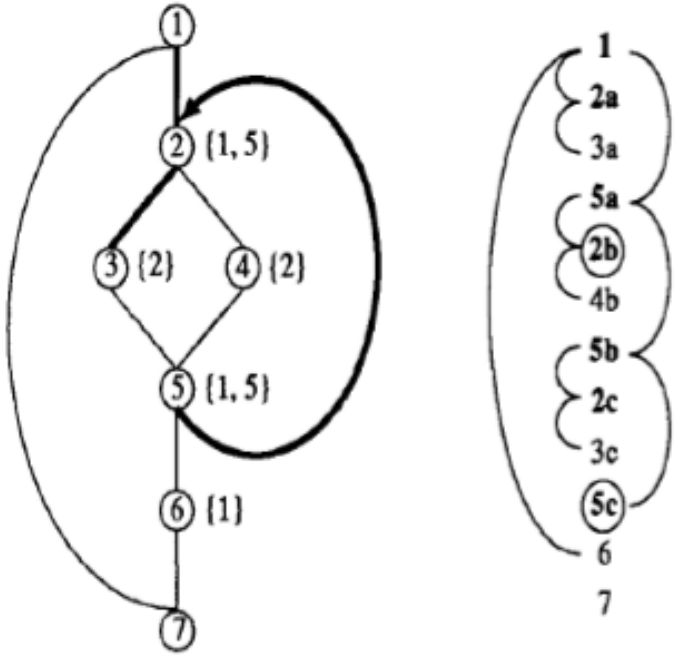
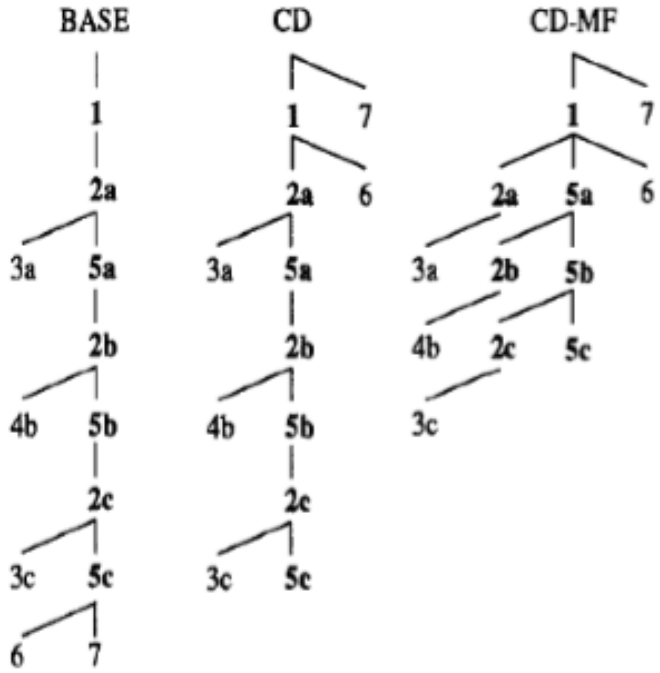


Figure 2: Example Flow Graph and Trace



CD-MF An instruction cannot execute until its control dependence branches are resolved. Multiple branch instructions can execute in parallel and need not be ordered.

CD An instruction cannot execute until its control dependence branches are resolved. In addition, branch instructions must execute in order, one per cycle, to reflect the inability to pursue multiple flows of control simultaneously.

Abstract Machine Models

SP An instruction cannot execute until the immediately preceding mispredicted branch in the trace is resolved. This implies that a branch instruction must wait for all preceding mispredicted branches.

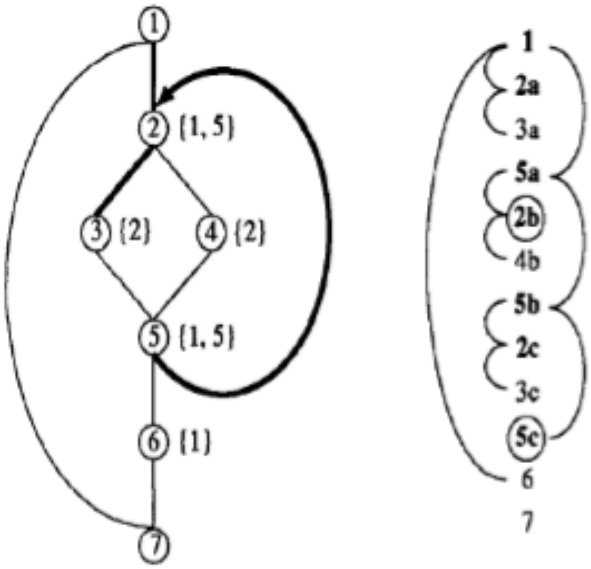
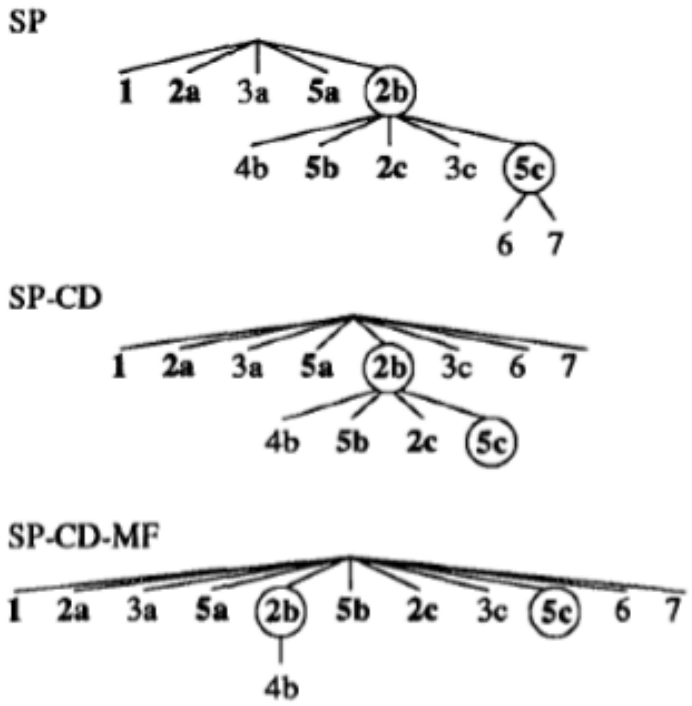


Figure 2: Example Flow Graph and Trace



SP-CD An instruction cannot execute until its mispredicted control dependence branches are resolved. In addition, a branch instruction must wait for all preceding mispredicted branches, not just the ones it is control dependent on.

SP-CD-MF An instruction cannot execute until its mispredicted control dependence branches are resolved. There are no additional constraints on branches.

Overall Parallelism Results for Numeric and Non-numeric Applications

	BASE	CD	CD-MF	SP	SP-CD	SP-CD-MF	ORACLE
awk	2.85	3.24	5.32	9.22	12.89	41.88	242.77
ccom	2.13	2.51	5.61	6.92	9.83	18.05	46.80
eqntott	1.98	2.05	5.21	6.40	18.09	225.90	3282.91
espresso	1.51	1.54	7.49	4.16	19.55	402.85	742.30
gcc (cc1)	2.10	2.55	14.63	7.76	13.18	66.29	174.50
irsim	2.31	2.66	11.89	8.40	15.82	45.86	265.42
latex	2.71	3.17	6.18	7.60	9.72	18.65	131.69
Harmonic Mean	2.14	2.39	6.96	6.80	13.27	39.62	158.26
matrix300	293	432	68324	36192	108575	180632	188470
spice2g6	2.14	2.29	16.80	8.11	25.28	196.76	843.60
tomcatv	22.23	42.77	3237	124	1881	3918	3918

Table 3: Parallelism for each Machine Model

- Parallelism for the CD machine is primarily limited by the constraint that branches must be executed in order.
- The constraints for the CD-MF machine only require that true data and control dependence be observed, the parallelism for this machine is a limit for all systems without speculative execution.

Results for Parallelism with Control Dependence Analysis

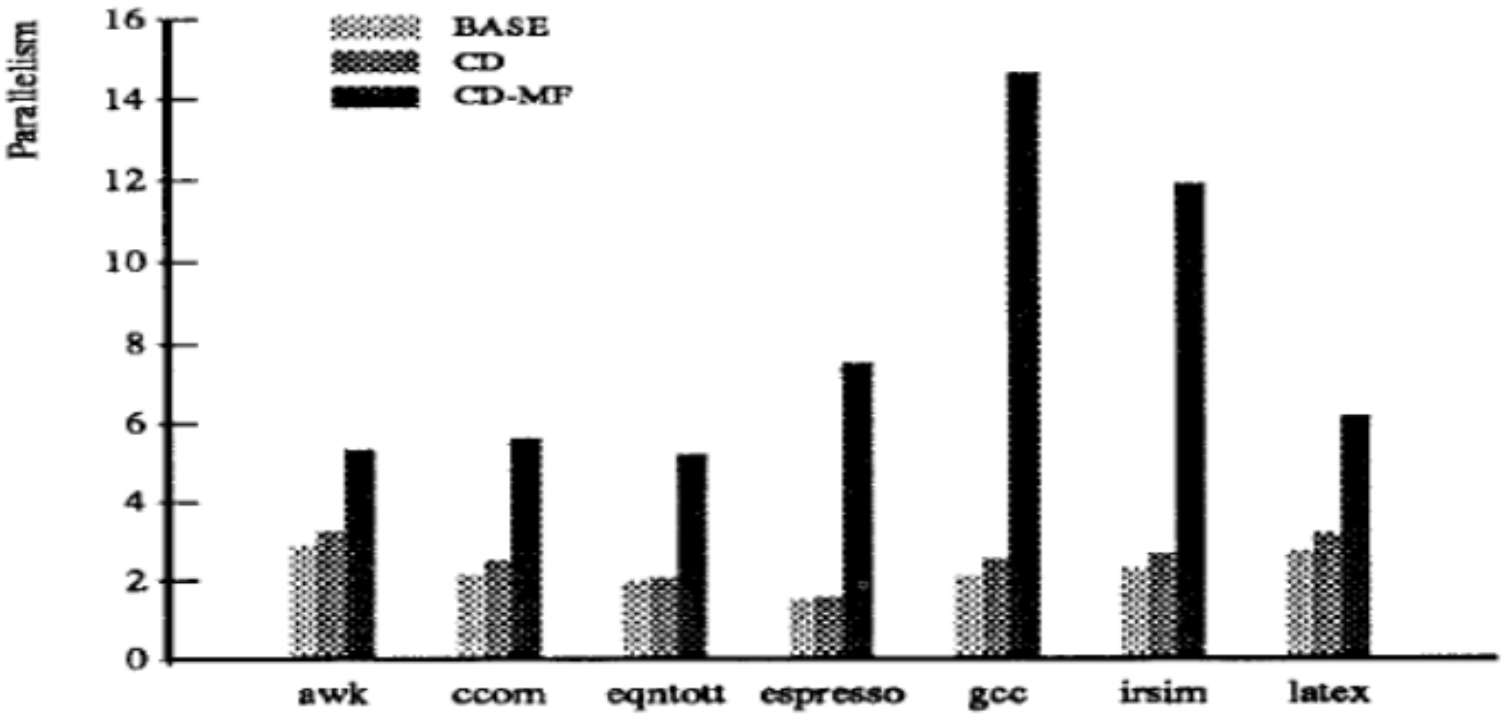


Figure 4: Parallelism with Control Dependence Analysis

- Parallelism for the CD machine is primarily limited by the constraint that branches must be executed in order.
- The constraints for the CD-MF machine only require that true data and control dependence be observed, the parallelism for this machine is a limit for all systems without speculative execution.

Results for Parallelism with Speculative Execution and Control Dependence Analysis

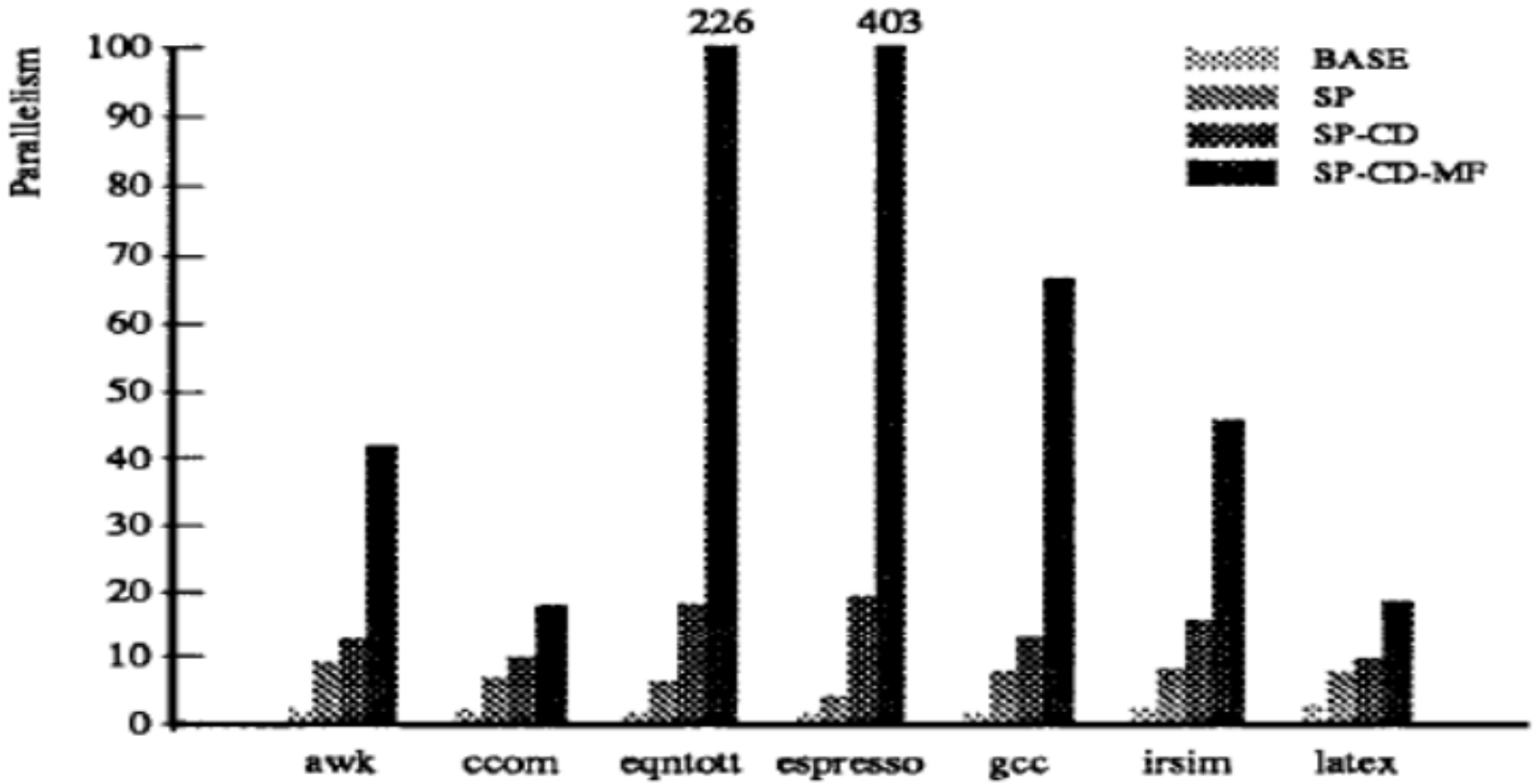


Figure 5: Parallelism with Speculative Execution

- These results are comparable to Wall's results for a similar machine

Conclusions

- Control flow in a program can severely limit the available parallelism
- To increase the available parallelism beyond the current level, the constraints imposed by control flow must be relaxed

- Some of the current highly parallel architectures lack adequate support for control flow

Automatically Characterizing Large Scale Program Behavior

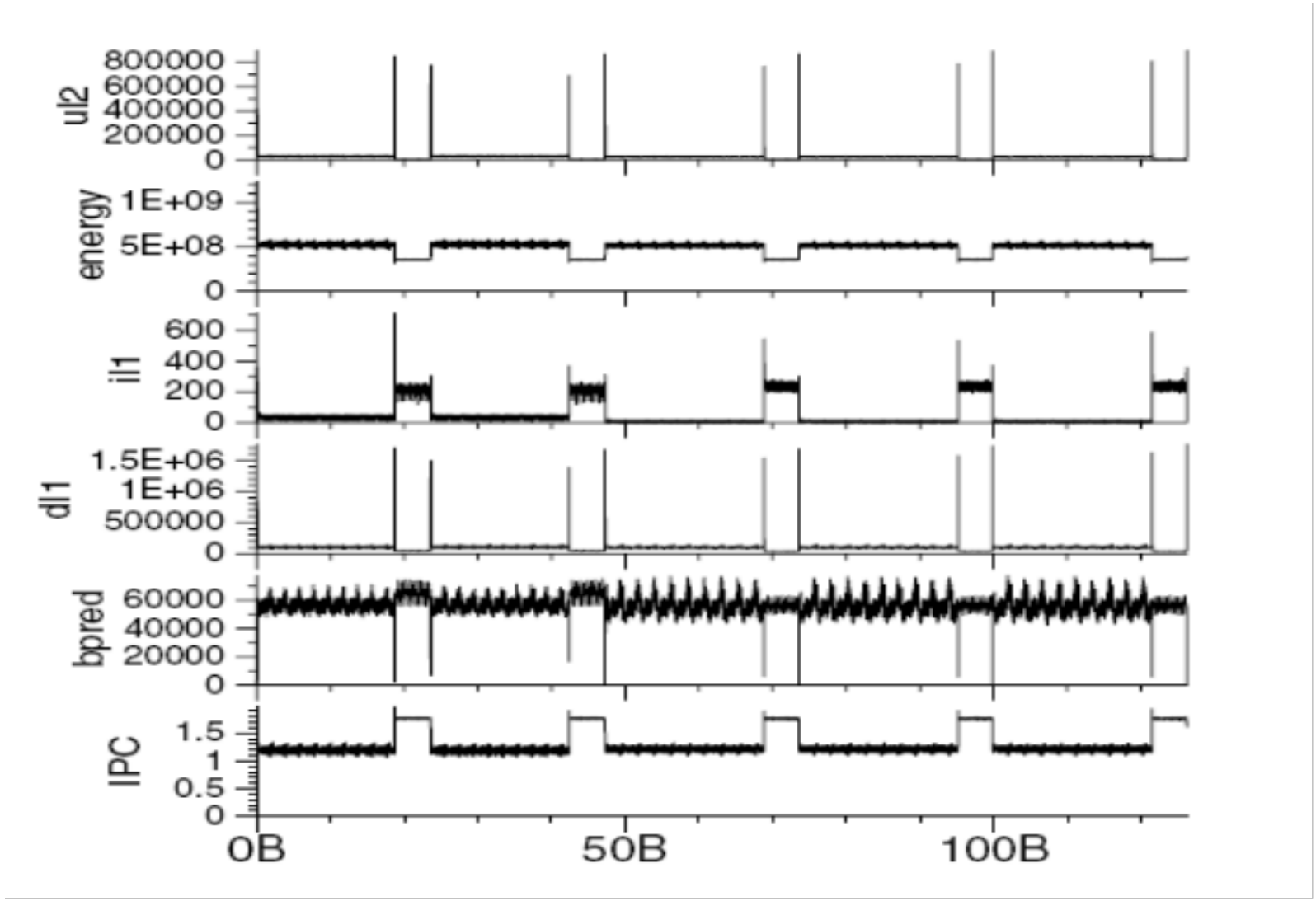
Author's Motivation and Goals

- **Motivation**: Programs can have wildly different behavior over their run time, and these behaviors can be seen even on the largest of scales. Understanding these large scale program behaviors can unlock many new optimizations

- **Goals**:
  - To create an automatic system that is capable of intelligently characterizing time-varying program behavior
  - To provide both analytic and software tools to help with program phase identification
  - To demonstrate the utility of these tools for finding places to simulate

- **How**: Develop a hardware independent metric that can concisely summarize the behavior of an arbitrary section of execution in a program

Large Scale Behavior for *gzip*



Approach

- Fingerprint each interval in program
  - Enables building the high level model
- Basic Block Vector
  - Tracks the code that is executing
  - Long sparse vector
  - Based on instruction execution frequency

Basic Block Vectors

BB	Assembly Code of bzip

1	srl a2, 0x8, t4
	and a2, 0xff, t12
	addl zero, t12, s6
	subl t7, 0x1, t7
	cmpeq s6, 0x25, v0
	cmpeq s6, 0, t0
	bis v0, t0, v0
	bne v0, 0x120018c48

2	subl t7, 0x1, t7
	cmple t7, 0x3, t2
	beq t2, 0x120018b04

3	ble t7, 0x120018bb4

4	and t4, 0xff, t5
	srl t4, 0x8, t4
	addl zero, t5, s6
	cmpeq s6, 0x25, s0
	cmpeq s6, 0, a0
	bis s0, a0, s0
	bne s0, 0x120018c48

5	subl t7, 0x1, t7 gt t7, 0x120018b90
...	...

For each interval:

ID: 1 2 3 4 5 .

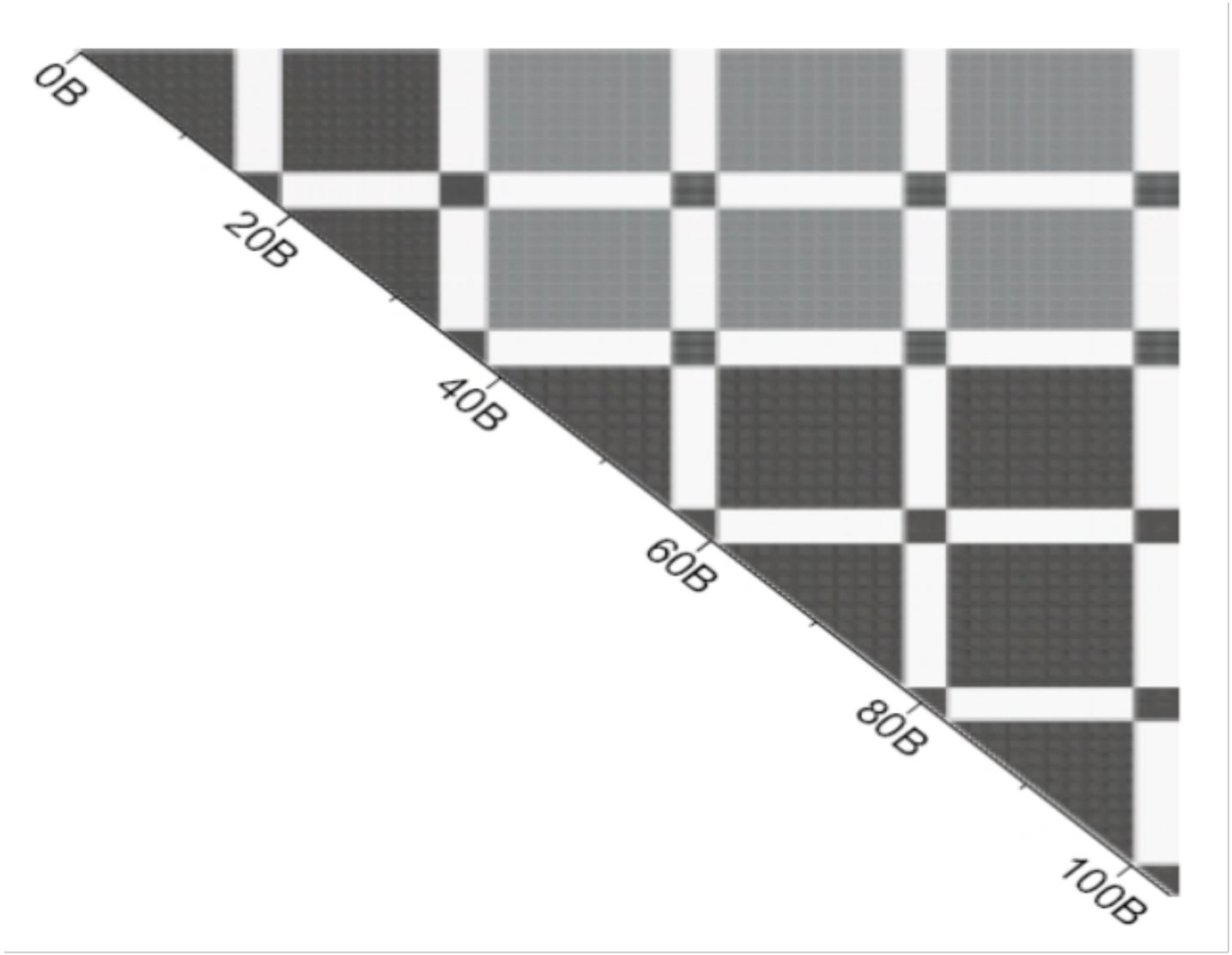
BB Exec Count: <1, 20, 0, 5, 0, …>

weigh by Block Size: <8, 3, 1, 7, 2, …>

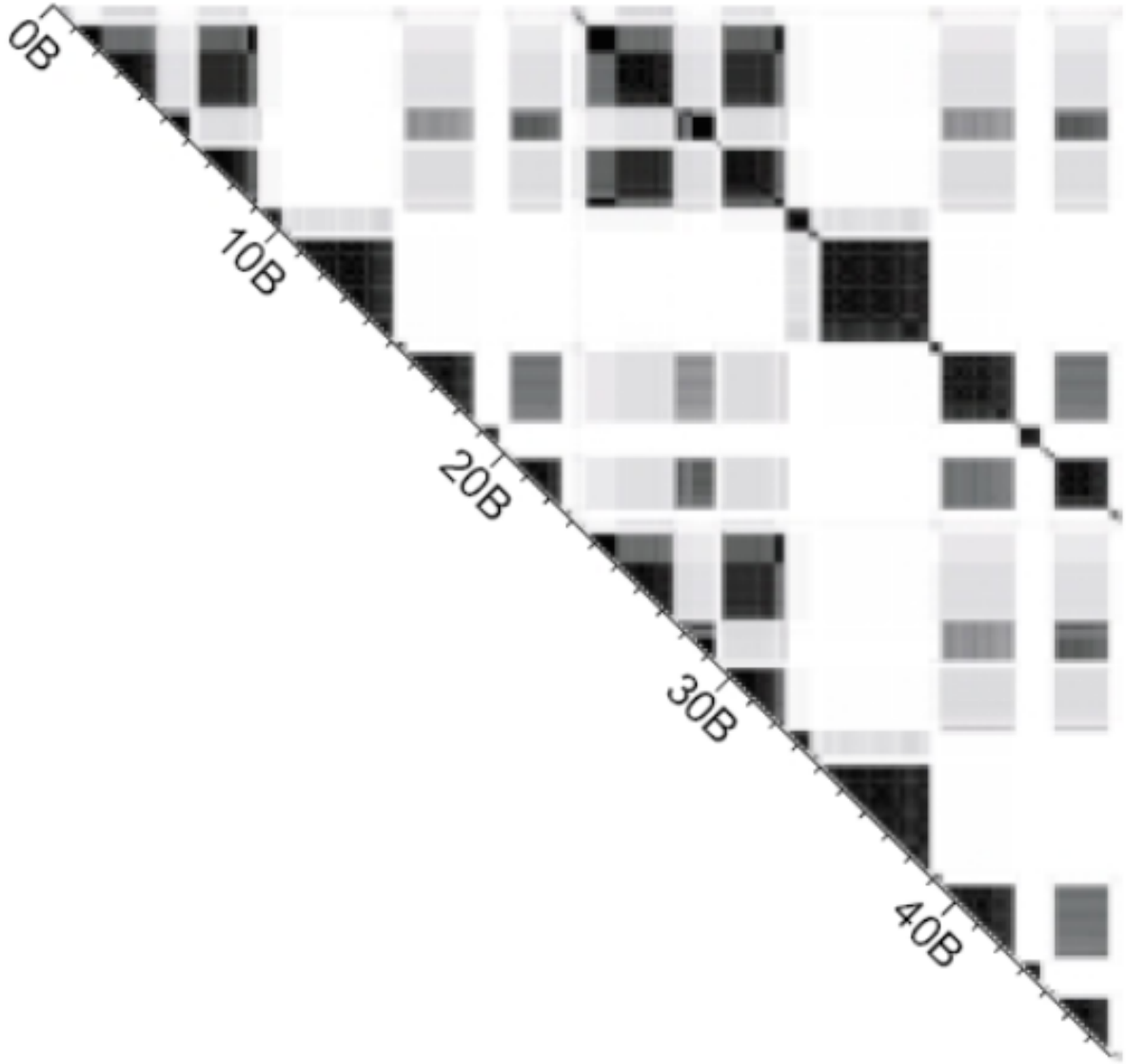
= <8, 60, 0, 35, 0, …>

Normalize to 1 = <8%,58%,0%,34%,0%,…>

Basic Block Similarity Matrix for *gzip*



Basic Block Similarity Matrix for *gcc*



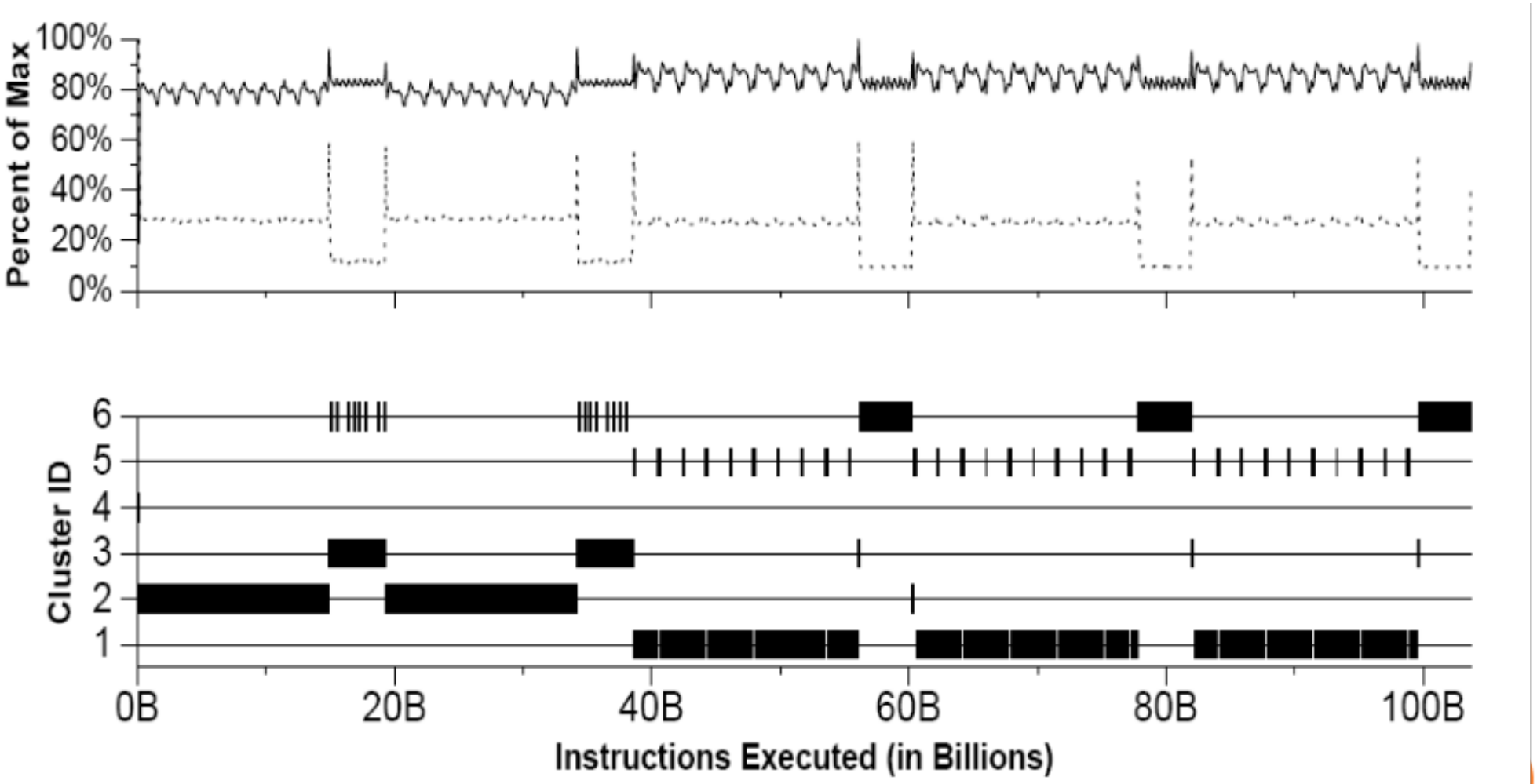
Finding the Phases

- Basic Block Vector provide a compact and representative summary of the program's behavior for intervals of execution
- But need to start by delineating a method of finding and representing the information; because there are so many intervals of execution that are similar to one another, one efficient representation is to group the intervals together that have similar behavior
- This problem is analogous to a *clustering* problem
- A Phase is a Cluster of BBVectors

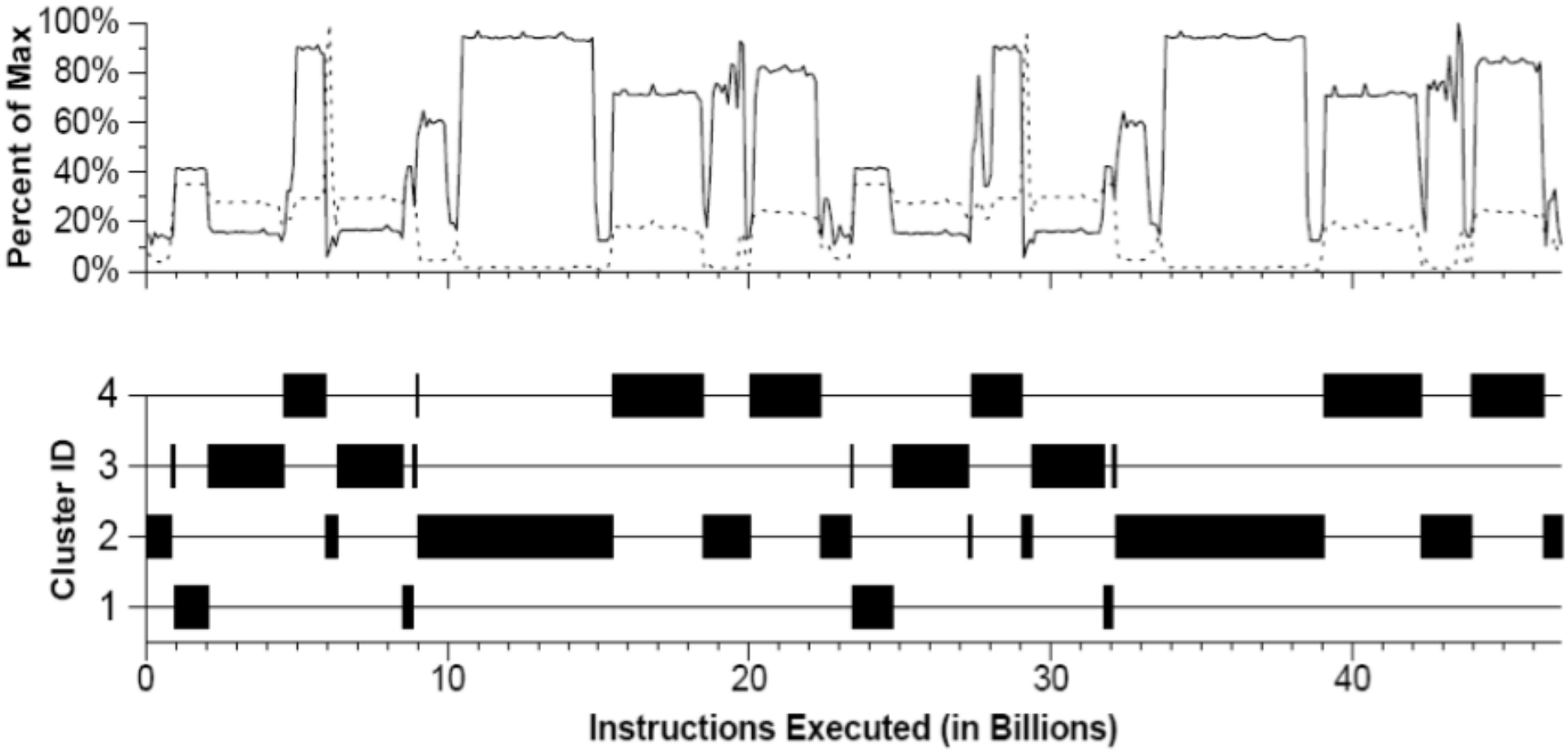
Phase-finding Algorithm

1. Profile the basic blocks
2. Reduce the dimension of the BBV data to 15 dimensions using random linear projection
3. Use the k -means algorithm to find clusters in the data for many different values of k
4. Choose the clustering with the smallest k , such that it's score is at least 90% as good as the best score.

Time varying and Cluster graph for *gzip*



Time varying and Cluster graph for *gcc*



Applications: Efficient Simulation

- Simulating to completion not feasible
  - Detailed simulation on SPEC takes months
  - Cycle level effects can't be ignored

- To reduce simulation time it is only feasible to execute a small portion of the program, it is very important that the section simulated is an accurate representation of the program's behavior as a whole.

- A *SimPoint* is a starting simulation place in a program's execution derived from their analysis.

Results for Single *SimPoint*

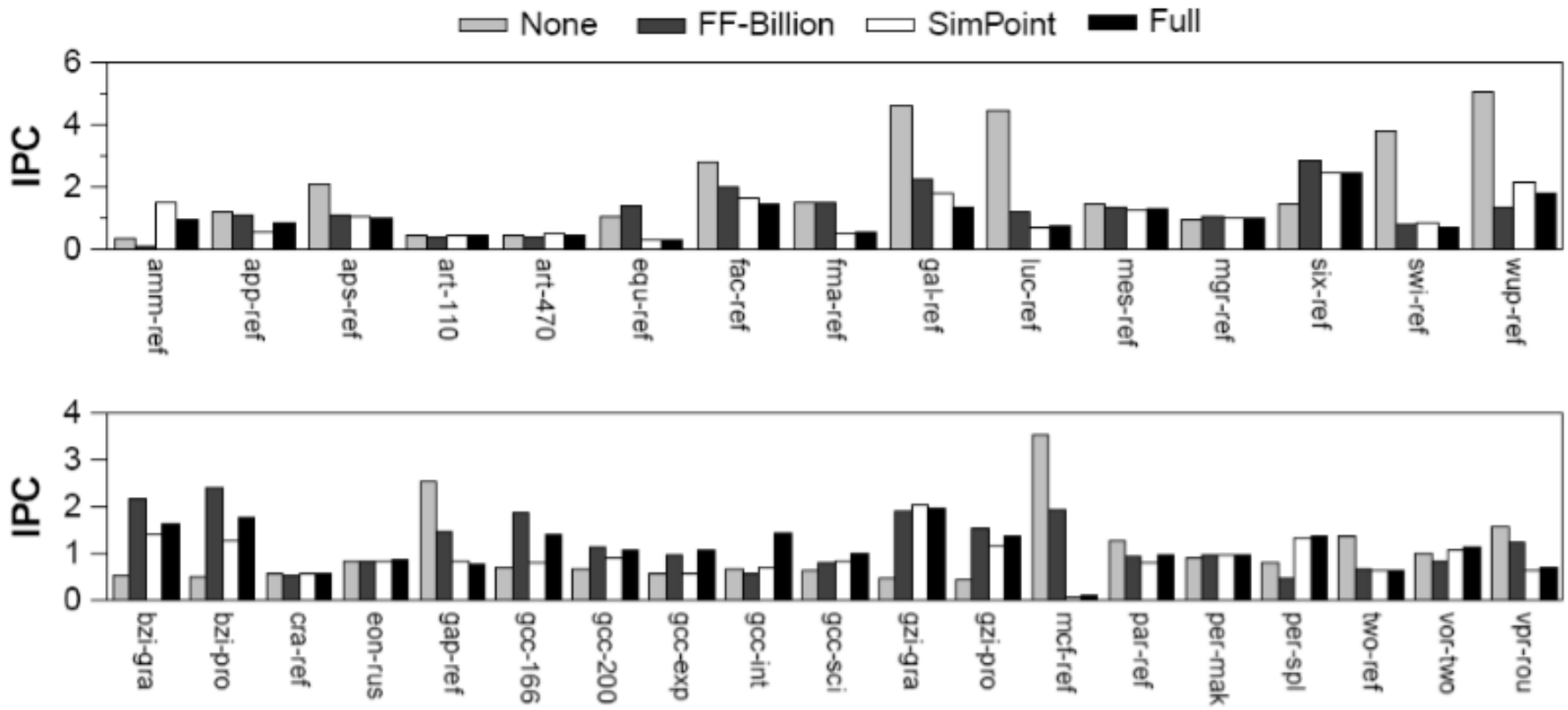


Figure 9: Simulation results starting simulation at the start of the program (none), blindly fast forwarding 1 billion instructions, using a single simulation point, and the IPC of the full execution of the program.

Multiple *SimPoints*

- Perform phase analysis
- For each phase in the program
  - Pick the interval most representative of the phase
  - This is the *SimPoint* for that phase
- Perform detailed simulation for *SimPoints*
- Weigh results for each *SimPoint*
  - According to the size of the phase it represents

Results for Multiple *SimPoints*

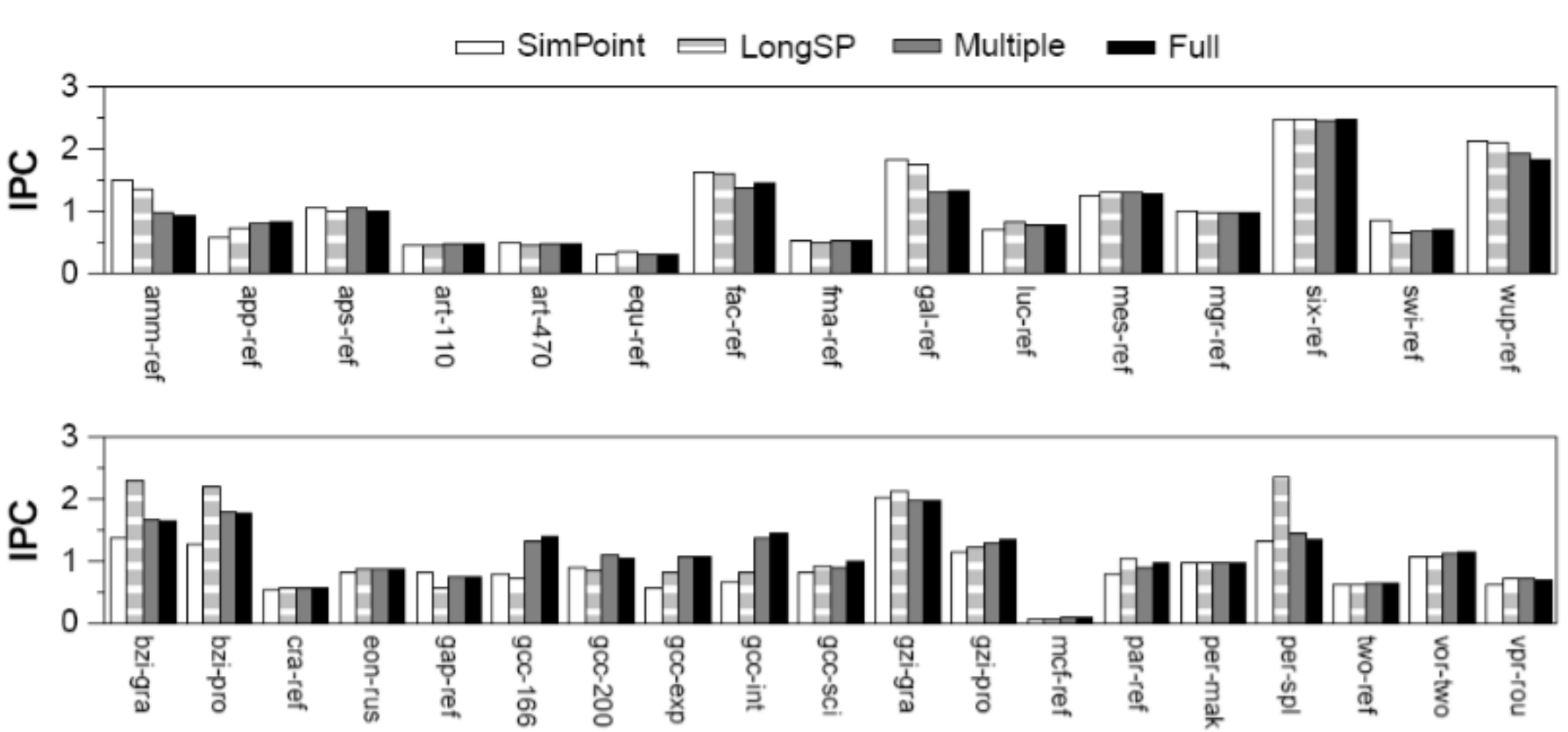


Figure 10: Multiple simulation point results. Simulation results are shown for using a single simulation point simulating for 100 million instructions, LongSP chooses a single simulation point simulating for the same length of execution as the multiple point simulation, simulation using multiple simulation points, and the full execution of the program.

Conclusions

- Use their analytical tools to automatically and efficiently analyze program behavior over large sections of execution, in order to take advantage of structure found in the program

- Clustering analysis can be used to automatically find multiple simulation points to reduce simulation time and to accurately model full program behavior

Back-up Slides

Comparisons Between Different Architectures

	Sequential Architecture	**Dependence Architecture**	**Independence Architectures**
Additional info required in the program	None	Specification of dependences between operations	Minimally, a partial list of independences. A complete specification of when and where each operation to be executed
Typical kind of ILP processor	Superscalar	Dataflow	VLIW
Dependences analysis	Performed by HW	Performed by compiler	Performed by compiler
Independences analysis	Performed by HW	Performed by HW	Performed by compiler
Scheduling	Performed by HW	Performed by HW	Performed by compiler
Role of compiler	Rearranges the code to make the analysis and scheduling HW more successful	Replaces some analysis HW	Replaces virtually all the analysis and scheduling HW

Large Scale Behavior for *gcc*