LoopPoint: Checkpoint-driven Sampled Simulation for Multi-threaded Applications

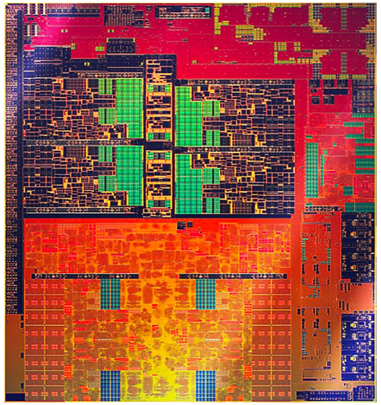
Alen Sabu[1], Harish Patil[2], Wim Heirman[2], Trevor E. Carlson[1]

[1]National University of Singapore

[2]Intel Corporation

Simulation in the Post-Dennard Era

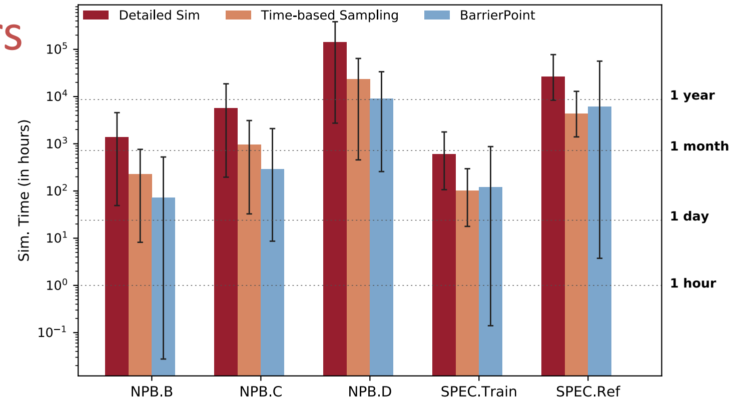
- Modern architectures require smarter simulators



Intel's 10nm Ice Lake die shot.
Image source: Intel

Simulation in the Post-Dennard Era

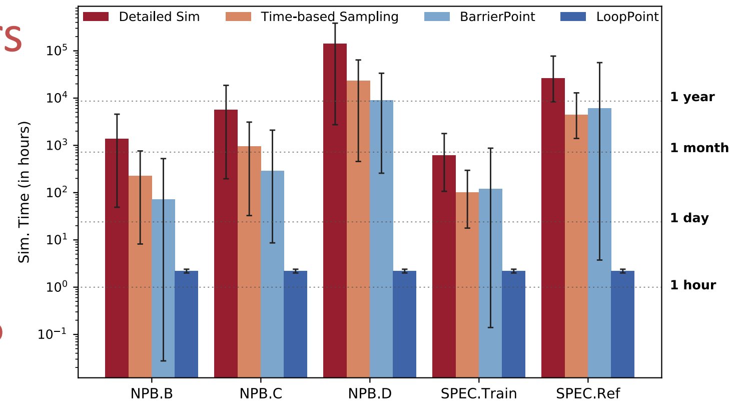
- Modern architectures require smarter simulators
- Microarchitectural simulation is slow
  - NPB (D), SPEC CPU2017 (ref) can take years
  - Solution – Simulate representative sample



Benchmarks with *8* threads, *static* schedule, *passive* wait-policy, simulated at *100 KIPS*.

Simulation in the Post-Dennard Era

- Modern architectures require smarter simulators
- Microarchitectural simulation is slow
  - NPB (D), SPEC CPU2017 (ref) can take years
  - Solution – Simulate representative sample
- Can we further bring down the simulation time?



Benchmarks with *8* threads, *static* schedule, *passive* wait-policy, simulated at *100 KIPS*.

Multi-threaded Sampling is Complex

- Instruction count-based techniques are unsuitable[1]
  - Forces determinism among threads
  - Ignoring spinloops – inflates IPC

[1]Alameldeen et al., “IPC Considered Harmful for Multiprocessor Workloads”, IEEE Micro 2006

Multi-threaded Sampling is Complex

- Instruction count-based techniques are unsuitable[1]
  - Forces determinism among threads
  - Ignoring spinloops – inflates IPC
- Threads progress differently due to load imbalance
  - Locks are acquired in different order
  - Unprotected shared memory accesses differ

[1]Alameldeen et al., “IPC Considered Harmful for Multiprocessor Workloads”, IEEE Micro 2006

Multi-threaded Sampling is Complex

- Instruction count-based techniques are unsuitable[1]
  - Forces determinism among threads
  - Ignoring spinloops – inflates IPC
  - Unprotected shared memory accesses differ

Identify a *unit of work* that is *invariant* across executions

[1]Alameldeen et al., "IPC Considered Harmful for Multiprocessor Workloads", IEEE Micro 2006

Application Sampling In-place

Single-threaded Sampling

SimPoint[1]
SMARTS[2] → Instruction count

Multiprocessor Sampling

Flex Points[3] → Instruction count

Multi-threaded Sampling

Time-based sampling[4] → Time

Application-specific Sampling

BarrierPoint[5] → Inter-barrier regions
TaskPoint[6] → Task instances

[1]Sherwood et al., "Automatically Characterizing Large Scale Program Behavior", ASPLOS'02

[2]Wunderlich et al., "SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling", ISCA'03

Application Sampling In-place

Single-threaded Sampling

SimPoint[1]

SMARTS[2]

$\rightarrow$ Instruction count

Multiprocessor Sampling

Flex Points[3] $\rightarrow$ Instruction count

Multi-threaded Sampling

Time-based sampling[4] $\rightarrow$ Time

Application-specific Sampling

BarrierPoint[5] $\rightarrow$ Inter-barrier regions

TaskPoint[6] $\rightarrow$ Task instances

[1]Sherwood et al., “Automatically Characterizing Large Scale Program Behavior”, ASPLOS’02

[2]Wunderlich et al., “SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling”, ISCA’03

[3]Wenisch et al., “SimFlex: statistical sampling of computer system simulation”, IEEE Micro’06

Application Sampling In-place

Single-threaded Sampling

SimPoint[1]
SMARTS[2] → Instruction count

Multiprocessor Sampling

Flex Points[3] → Instruction count

Multi-threaded Sampling

Time-based sampling[4] → Time

Application-specific Sampling

BarrierPoint[5] → Inter-barrier regions
TaskPoint[6] → Task instances

[1]Sherwood et al., "Automatically Characterizing Large Scale Program Behavior", ASPLOS'02
[2]Wunderlich et al., "SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling", ISCA'03
[3]Wenisch et al., "SimFlex: statistical sampling of computer system simulation", IEEE Micro'06
[4]Carlson et al., "Sampled Simulation of Multithreaded Applications", ISPASS'13

Application Sampling In-place

Single-threaded Sampling

SimPoint[1]
SMARTS[2]
$\rightarrow$ Instruction count

Multiprocessor Sampling

Flex Points[3] $\rightarrow$ Instruction count

Multi-threaded Sampling

Time-based sampling[4] $\rightarrow$ Time

Application-specific Sampling

BarrierPoint[5] $\rightarrow$ Inter-barrier regions

TaskPoint[6] $\rightarrow$ Task instances

[1]Sherwood et al., "Automatically Characterizing Large Scale Program Behavior", ASPLOS'02

[2]Wunderlich et al., "SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling", ISCA'03

[3]Wenisch et al., "SimFlex: statistical sampling of computer system simulation", IEEE Micro'06

[4]Carlson et al., "Sampled Simulation of Multithreaded Applications", ISPASS'13

[5]Carlson et al., "BarrierPoint: Sampled simulation of multi-threaded applications", ISPASS'14

[6]Grass et al., "TaskPoint: Sampled simulation of task-based programs", ISPASS'16

Application Sampling In-place

Single-threaded Sampling

SimPoint[1]
SMARTS[2] → Instruction count

Time-based sampling[4] → Time

Multiprocessor Sampling

Flex Points[3] → Instruction count

BarrierPoint[5] → Inter-barrier regions

TaskPoint[6] → Task instances

We consider the number of loop executions as unit of work

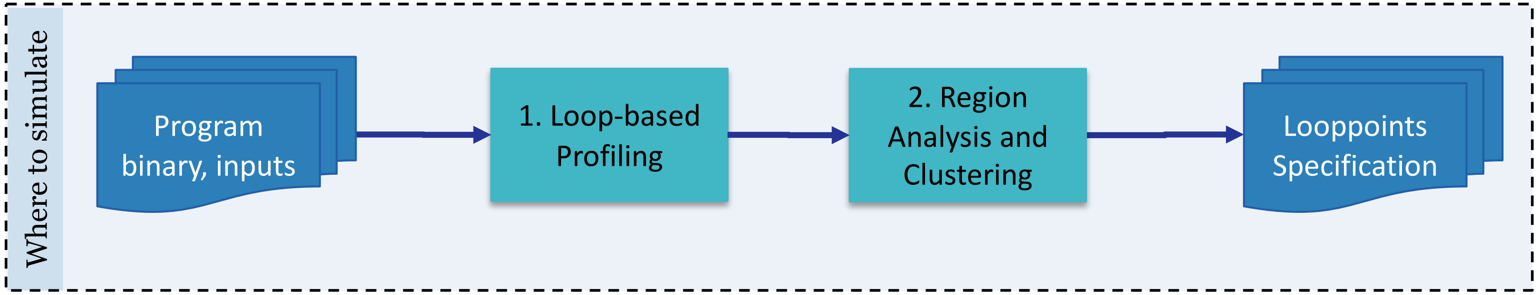
[1]Sherwood et al., “Automatically Characterizing Large Scale Program Behavior”, ASPLOS’02
[2]Wunderlich et al., “SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling”, ISCA’03
[3]Wenisch et al., “SimFlex: statistical sampling of computer system simulation”, IEEE Micro’06
[4]Carlson et al., “Sampled Simulation of Multithreaded Applications”, ISPASS’13
[5]Carlson et al., ”BarrierPoint: Sampled simulation of multi-threaded applications”, ISPASS’14
[6]Grass et al., “TaskPoint: Sampled simulation of task-based programs”, ISPASS’16

Overall Methodology

Where to simulate

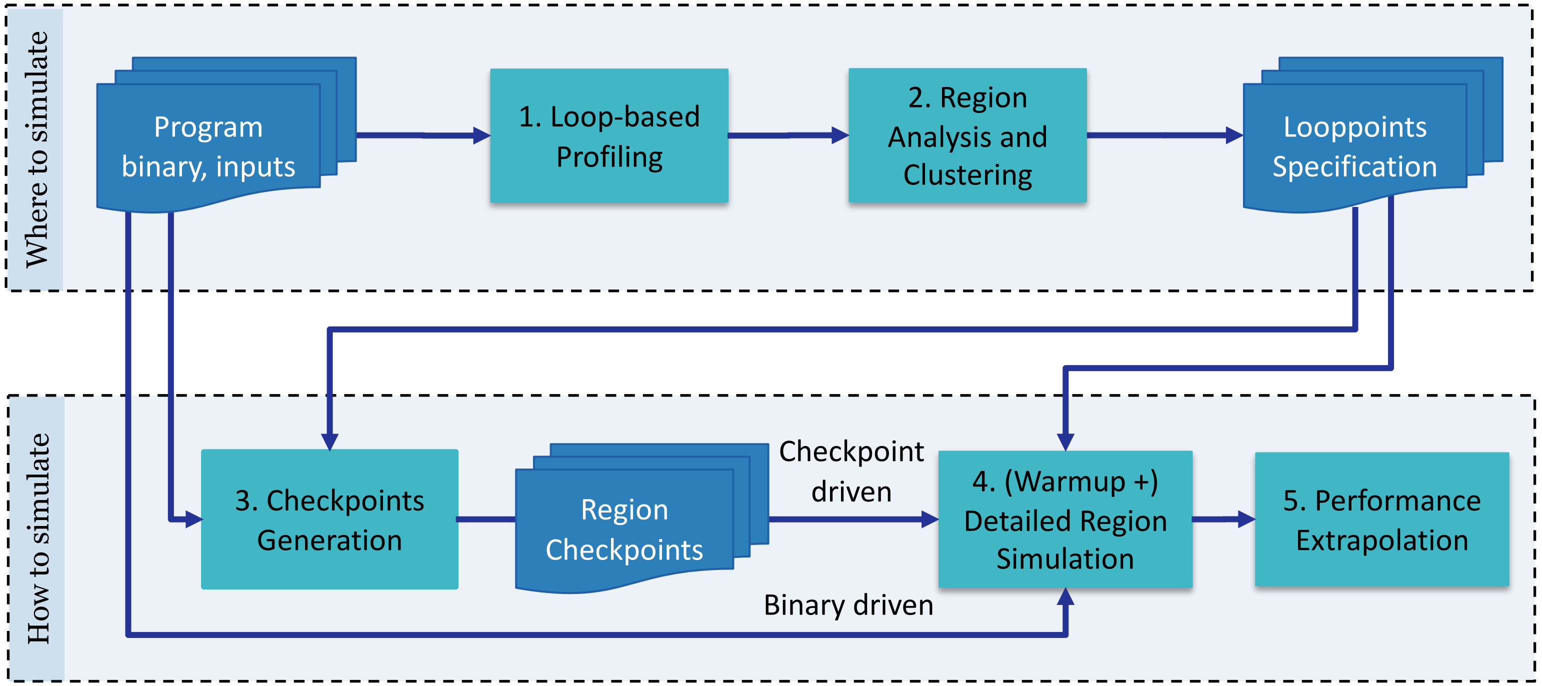
How to simulate

Overall Methodology

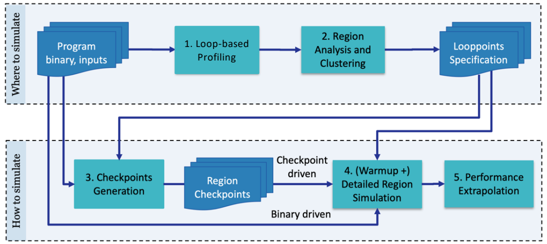


How to simulate

Overall Methodology

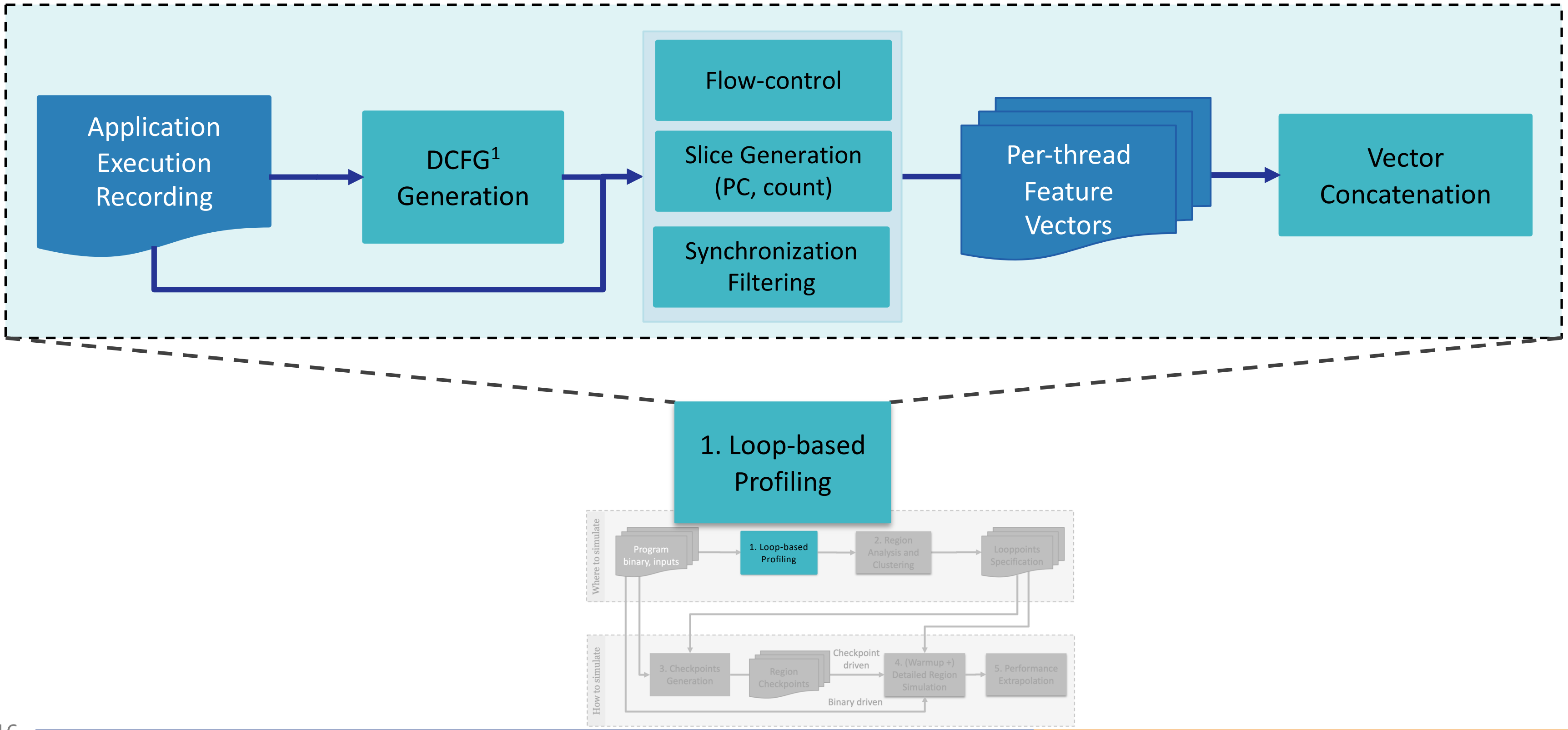


Loop-based Profiling



[1]DCFG: Dynamic Control-Flow Graph

Loop-based Profiling

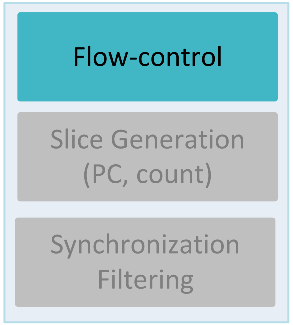


Application Execution Recording
DCFG[1] Generation
Flow-control
Slice Generation (PC, count)
Synchronization Filtering
Per-thread Feature Vectors
Vector Concatenation
1. Loop-based Profiling
Where to simulate
Program binary, inputs
1. Loop-based Profiling
2. Region Analysis and Clustering
Looppoints Specification
How to simulate
3. Checkpoints Generation
Region Checkpoints
Checkpoint driven
4. (Warmup +) Detailed Region Simulation
5. Performance Extrapolation
Binary driven

[1]DCFG: Dynamic Control-Flow Graph

Loop-based Profiling: Flow-control

- Load Imbalance can affect profiling
  - Make sure threads make equal forward progress
- Implementation: Control the forward progress of threads
  - Synchronize threads (barriers) externally at regular intervals
  - Make sure all threads execute similar number of instructions



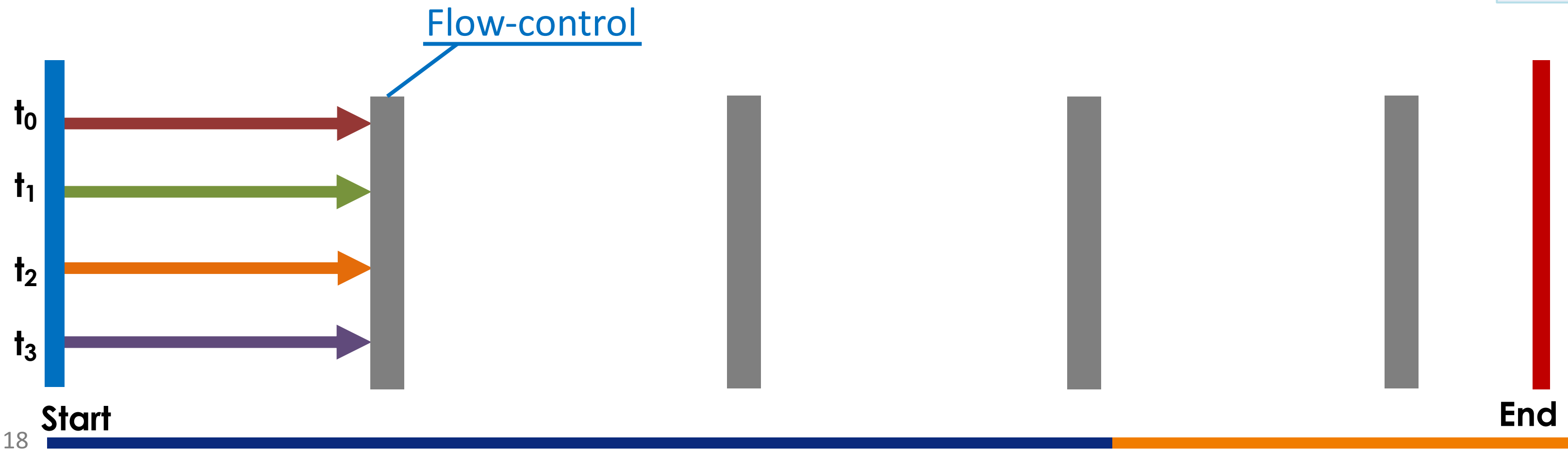
Loop-based Profiling: Flow-control

- Load Imbalance can affect profiling
  - Make sure threads make equal forward progress
- Implementation: Control the forward progress of threads
  - Synchronize threads (barriers) externally at regular intervals
  - Make sure all threads execute similar number of instructions

Flow-control

Slice Generation (PC, count)

Synchronization Filtering



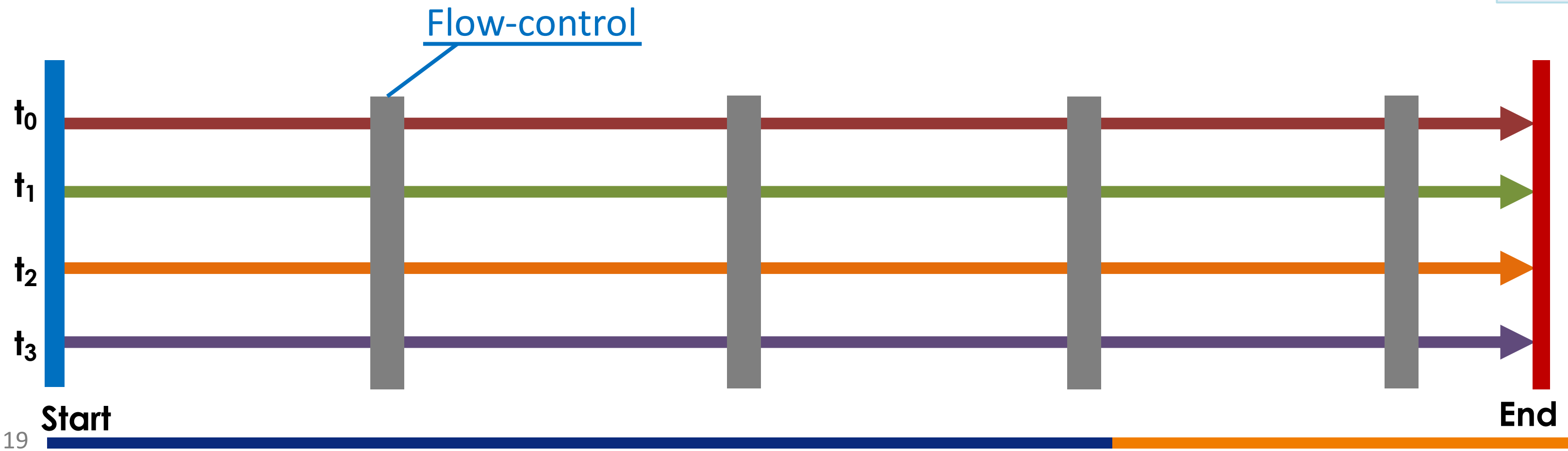
Loop-based Profiling: Flow-control

- Load Imbalance can affect profiling
  - Make sure threads make equal forward progress
- Implementation: Control the forward progress of threads
  - Synchronize threads (barriers) externally at regular intervals
  - Make sure all threads execute similar number of instructions

Flow-control

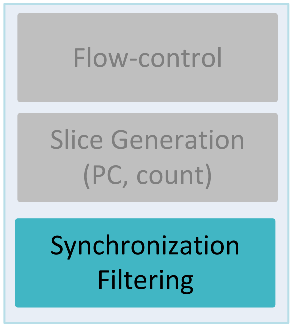
Slice Generation (PC, count)

Synchronization Filtering



Loop-based Profiling: Sync Filtering

- Goal: Filter out synchronization during profiling
  - Profiling data should contain only ***real*** work
- Solutions:
  - Automatic detection using loop analysis[1]
  - Ignore sync library code (Ex. `libiomp5.so`, `libpthread.so`)



[1]Li et al., "Spin detection hardware for improved management of multithreaded systems," TPDS, 2006

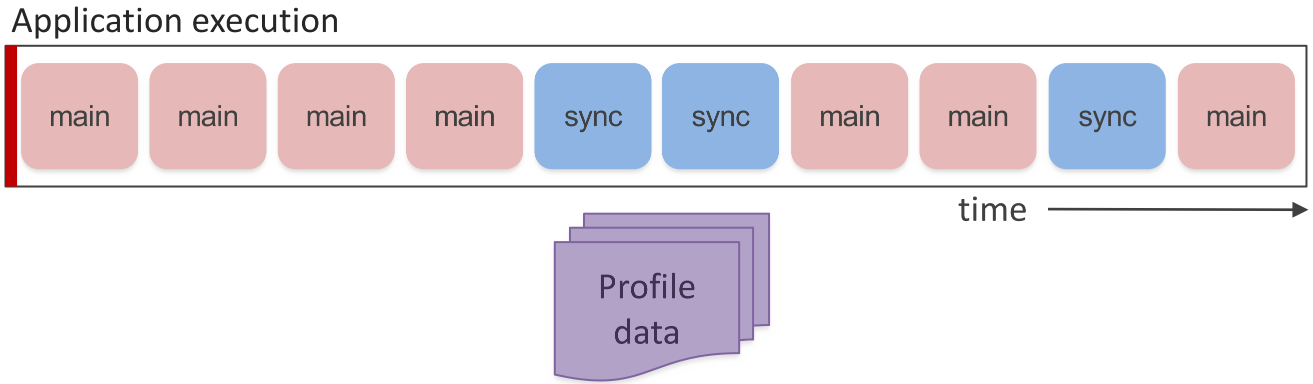
Loop-based Profiling: Sync Filtering

- Goal: Filter out synchronization during profiling
  - Profiling data should contain only ***real*** work
- Solutions:
  - Automatic detection using loop analysis[1]
  - Ignore sync library code (Ex. `libiomp5.so`, `libpthread.so`)

Flow-control

Slice Generation (PC, count)

Synchronization Filtering



[1]Li et al., "Spin detection hardware for improved management of multithreaded systems," TPDS, 2006

Loop-based Profiling: Slice Generation

- Region start/stop
  - Global instruction count reaches threshold (*#threads* × 100 M)
  - Region boundary at a loop entry/exit – use DCFG analysis
- Looppoint region markers ***(PC, count$_{PC}$)***
  - Global count of loop entries: invariant across executions
  - Simulate the same ***amount of work***

Flow-control

Slice Generation
(PC, count)

Synchronization
Filtering

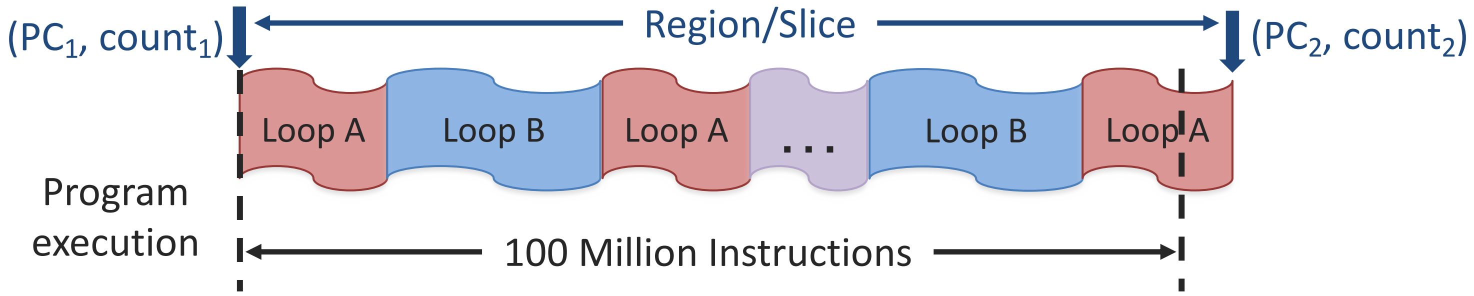
Loop-based Profiling: Slice Generation

- Region start/stop
  - Global instruction count reaches threshold (*#threads* × 100 M)
  - Region boundary at a loop entry/exit – use DCFG analysis
- Looppoint region markers ***(PC, count$_{PC}$)***
  - Global count of loop entries: invariant across executions
  - Simulate the same ***amount of work***

Flow-control

Slice Generation (PC, count)

Synchronization Filtering



Loop-based Profiling: Slice Generation

- Basic Block (BB)
  - Section of code with single entry and exit
- Basic Block Vector (BBV)
  - Execution fingerprint of an application interval
  - Vector with one element for each basic block BBV
  - Exec Wt = *entry count × number of instructions*

BB	Example Assembly Code
A	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
B	subl t7, 0x1, t7 cmple t7, 0x3, t2 beq t2, 0x120018b04
C	ble t7, 0x120018bb4
…	…

```
                 ID:    A   B   C
      BB Exec Count: < 1, 20, 0, …>
weigh by Block Size: < 8,  3, 1, …>
         BB Exec Wt: < 8, 60, 0, …>
```

Image source: Sherwood et.al., "Automatically Characterizing Large Scale Program Behavior", ASPLOS 2002

Loop-based Profiling: Slice Generation

- Basic Block (BB)
  - Section of code with single entry and exit
- Basic Block Vector (BBV)
  - Execu
  - Vecto
  - Exec Wt = *entry count × number of instructions*

```
[ A:8, B:60, C:0, …]
```

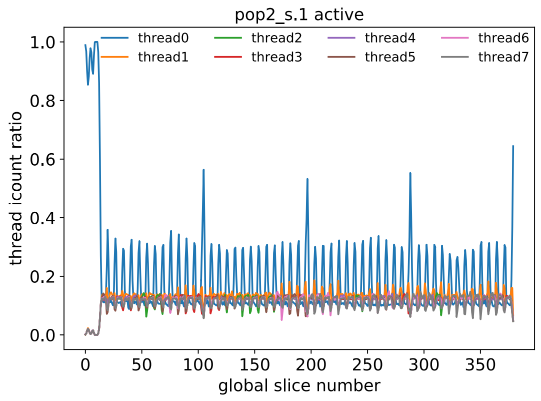
BB	Example Assembly Code
A	srl a2, 0x8, t4 and a2, 0xff, t12 addl zero, t12, s6 subl t7, 0x1, t7 cmpeq s6, 0x25, v0 cmpeq s6, 0, t0 bis v0, t0, v0 bne v0, 0x120018c48
B	subl t7, 0x1, t7 cmple t7, 0x3, t2 beq t2, 0x120018b04
C	ble t7, 0x120018bb4
…	…

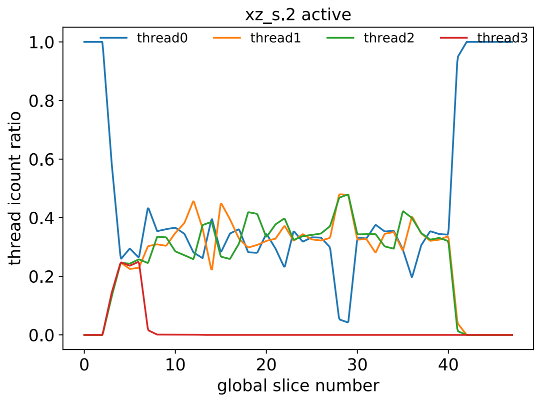
```
                  ID:   A   B   C
       BB Exec Count: < 1, 20, 0, …>
weigh by Block Size: < 8,  3, 1, …>
         BB Exec Wt: < 8, 60, 0, …>
```

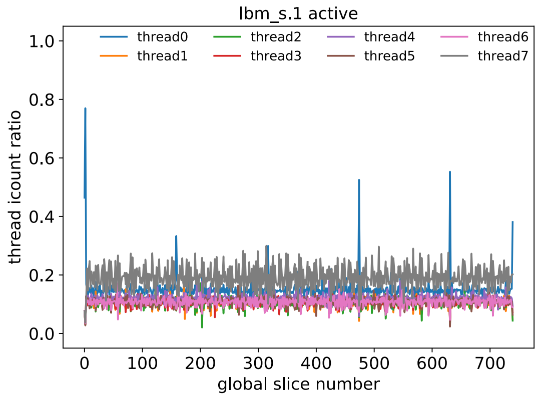
Image source: Sherwood et.al., "Automatically Characterizing Large Scale Program Behavior", ASPLOS 2002

Loop-based Profiling: Vector Concatenation

- Ratio of instructions per thread may differ







Loop-based Profiling: Vector Concatenation

- Ratio of instructions per thread may differ
- ***Global-BBV*s capture parallelism among threads**
  - Concatenate per-thread BBVs to larger Global BBV

BB	Example Assembly Code

A	`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`

B	`subl t7, 0x1, t7`
	`cmple t7, 0x3, t2`
	`beq t2, 0x120018b04`

C	`ble t7, 0x120018bb4`
…	…

M	`subl t7, 0x1, t7`
	`gt t7, 0x120018b90`

BB	Example Assembly Code

A	`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`

B	`subl t7, 0x1, t7`
	`cmple t7, 0x3, t2`
	`beq t2, 0x120018b04`

C	`ble t7, 0x120018bb4`
…	…

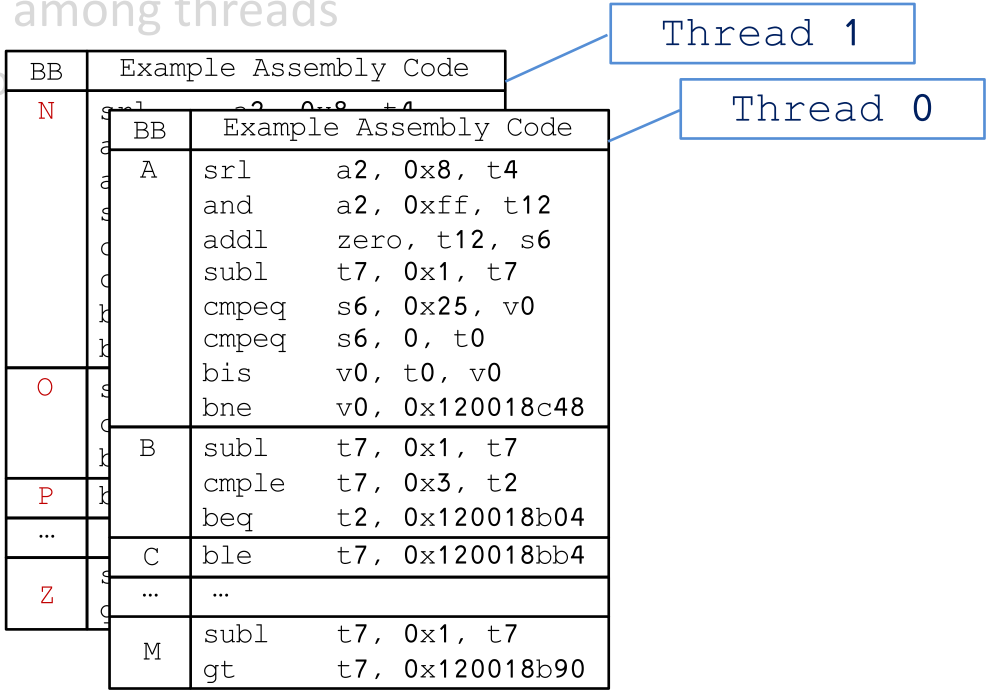
M	`subl t7, 0x1, t7`
	`gt t7, 0x120018b90`

Loop-based Profiling: Vector Concatenation

- Ratio of instructions per thread may differ
- *Global-BBVs* capture parallelism among threads
  - Concatenate per-thread BBVs to

BB ID:	**A**	**B**	**C**	…
BB Exec Wt:	< 8,	60,	0,	… >

BB ID:	**N**	**O**	**P**	…
BB Exec Wt:	< 8,	60,	0,	… >



BB	Example Assembly Code	
A	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
B	subl	t7, 0x1, t7
	cmple	t7, 0x3, t2
	beq	t2, 0x120018b04
C	ble	t7, 0x120018bb4
…	…	
M	subl	t7, 0x1, t7
	gt	t7, 0x120018b90

Loop-based Profiling: Vector Concatenation

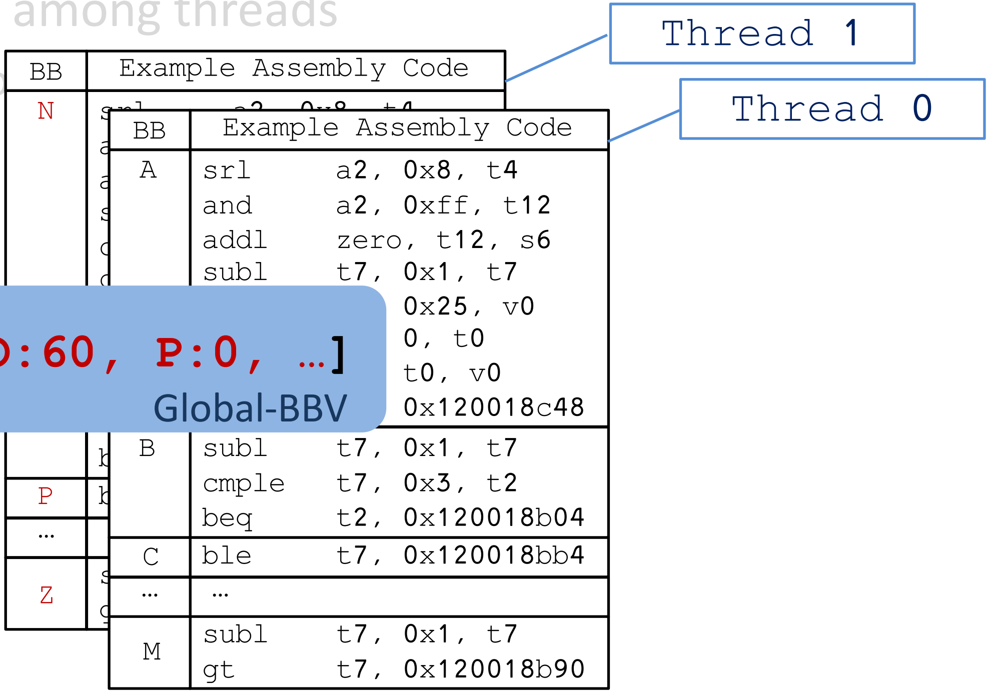
- Ratio of instructions per thread may differ
- *Global-BBV*s capture parallelism among threads
  - Concatenate per-thread BBVs to

BB ID:		A	B	C	…	
BB Exec Wt:	<	8,	60,	0,	…	>

[A:8, B:60, C:0, …, N:8, O:60, P:0, …]

Global-BBV

BB ID:		N	O	P	…	
BB Exec Wt:	<	8,	60,	0,	…	>


Thread 1
Thread 0

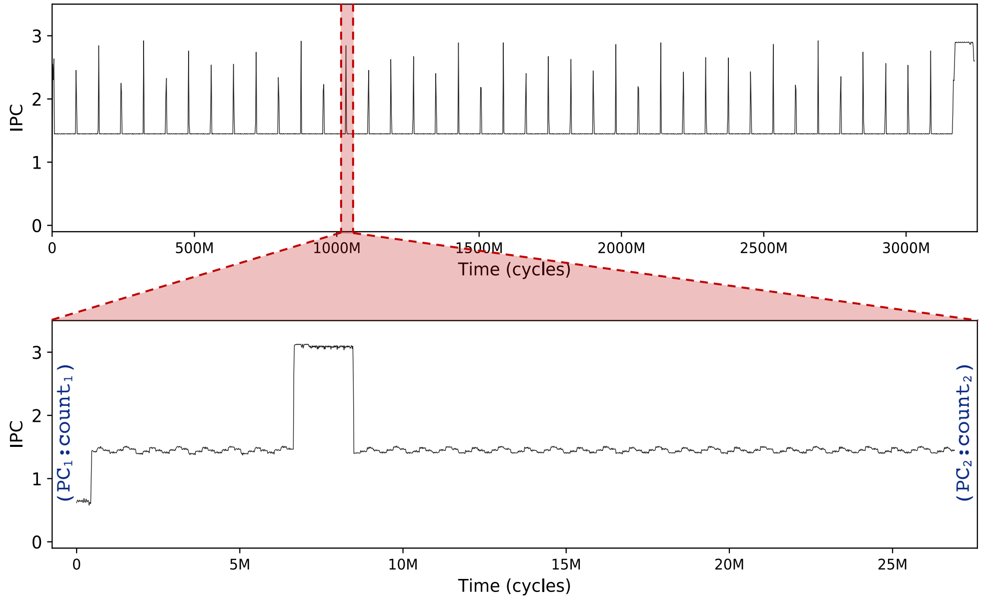
BB	Example Assembly Code	
N		
P		
…		
Z		

BB	Example Assembly Code	
A	srl	a2, 0x8, t4
	and	a2, 0xff, t12
	addl	zero, t12, s6
	subl	t7, 0x1, t7
		0x25, v0
		0, t0
		t0, v0
		0x120018c48
B	subl	t7, 0x1, t7
	cmple	t7, 0x3, t2
	beq	t2, 0x120018b04
C	ble	t7, 0x120018bb4
…	…	
M	subl	t7, 0x1, t7
	gt	t7, 0x120018b90

A LoopPoint Region

638.imagick_s/magick/morphology.c

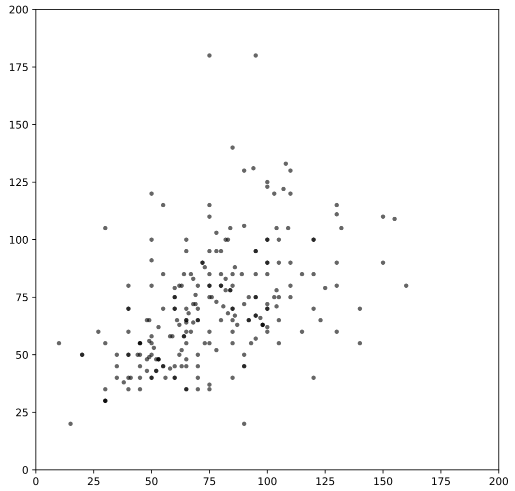
```
2842 #if defined(MAGICKCORE_OPENMP_SUPPORT)
2843   #pragma omp parallel for schedule(static,4) shared(progress,status) \
2844     magick_threads(image,result_image,image->rows,1)
2845 #endif
2846   for (y=0; y < (ssize_t) image->rows; y++)
2847   {
          ……
2886      for (x=0; x < (ssize_t) image->columns; x++)
2887      {
3021         for (v=0; v < (ssize_t) kernel->height; v++) {
3022           for (u=0; u < (ssize_t) kernel->width; u++, k--) {
          ……
3034           } /* u */
          ……
3037         }  /* v */
3342      } /* x */
3357   } /* y */
                ……
```



638.imagick_s, train input, 8 threads

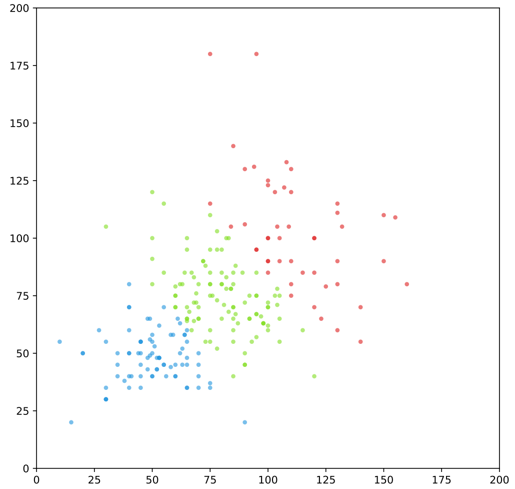
Identifying Simulation Regions

- Group similar Global-BBVs
  - K-means algorithm: Centroid-based clustering



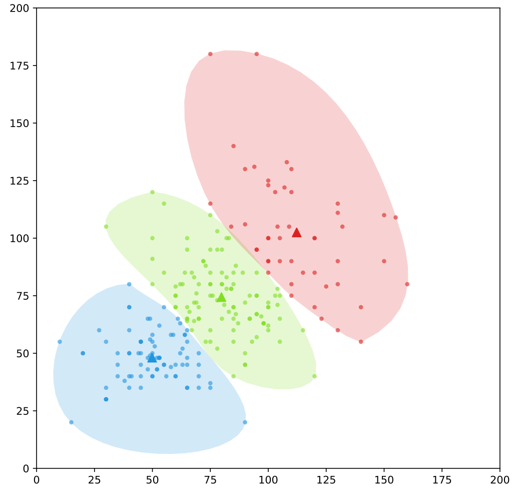
Identifying Simulation Regions

- Group similar Global-BBVs
  - K-means algorithm: Centroid-based clustering
- Vector closest to centroid is the representative



Identifying Simulation Regions

- Group similar Global-BBVs
  - K-means algorithm: Centroid-based clustering
- Vector closest to centroid is the representative
- Simulation regions (looppoints)
  - Checkpoints generated from the application
  - Use (PC, count_{PC}) information of representatives



Application Reconstruction

- Representative regions (looppoints) are simulated in parallel
- Warmup handling:
  - Simulate a large enough warmup region before simulation region
- Application performance
  - The weighted average of the performance of simulation regions

Application Reconstruction

- Representative regions (looppoints) are simulated in parallel
- Warmup handling:
  - Simulate a large enough warmup region before simulation region
- Application performance
  - The weighted average of the perform

$$multiplier_j = \frac{\sum_{i=0}^{m} inscount_i}{inscount_j}$$

$$\boldsymbol{total\,runtime = \sum_{i=rep_1}^{rep_N} runtime_i \times multiplier_i}$$

m regions represented by ***j***th looppoint

Experimental Setup

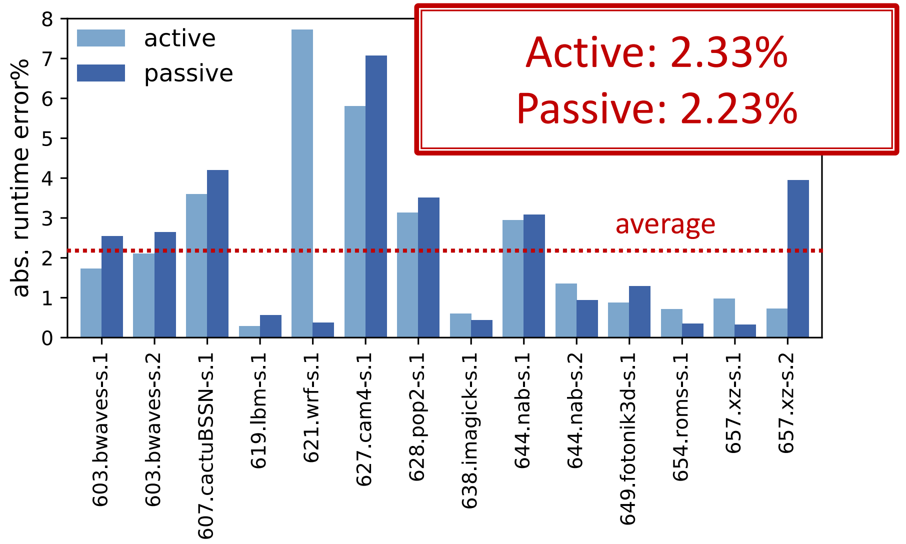
- Simulation Infrastructure
  - Sniper[1] 7.4
    - Mimics Intel Gainestown 8/16 core
- Benchmarks and OpenMP settings
  - SPEC CPU2017 speed benchmarks
    - Input: train; Threads: 8; Wait policy: Active, Passive
  - NAS Parallel Benchmarks (NPB)
    - Input: Class C; Threads: 8, 16; Wait policy: Passive
  - OpenMP scheduling policy: *static*

[1]Carlson et.al., "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation", SC 2011
Image source: https://www.openmp.org/; https://www.spec.org/cpu2017/ ; https://www.nas.nasa.gov/

Accuracy Results

- Prediction error wrt. performance of whole application

SPEC CPU2017 with train inputs, *8* threads

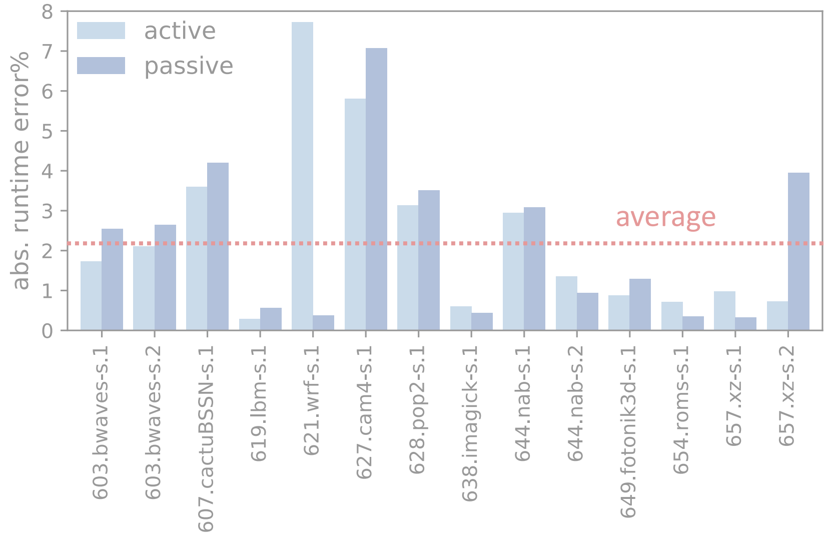


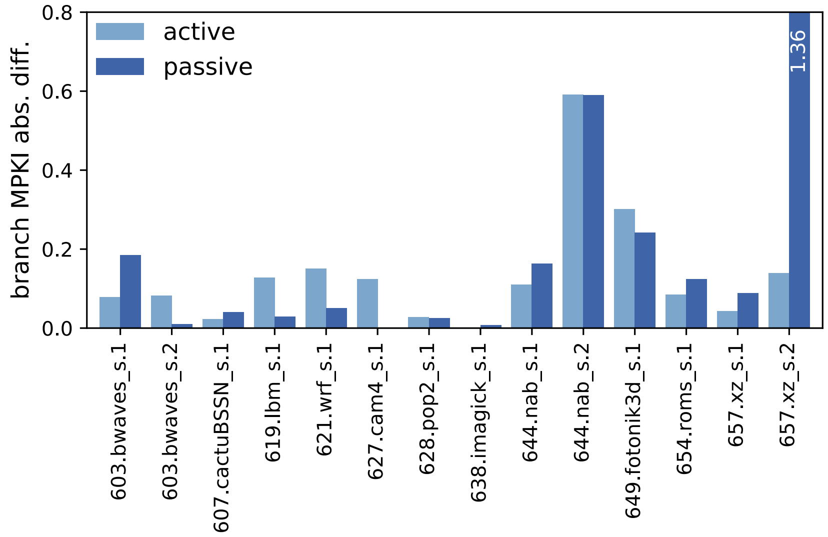
Active: 2.33%
Passive: 2.23%

Accuracy Results

- Prediction error wrt. performance of whole application

SPEC CPU2017 with train inputs, *8* threads

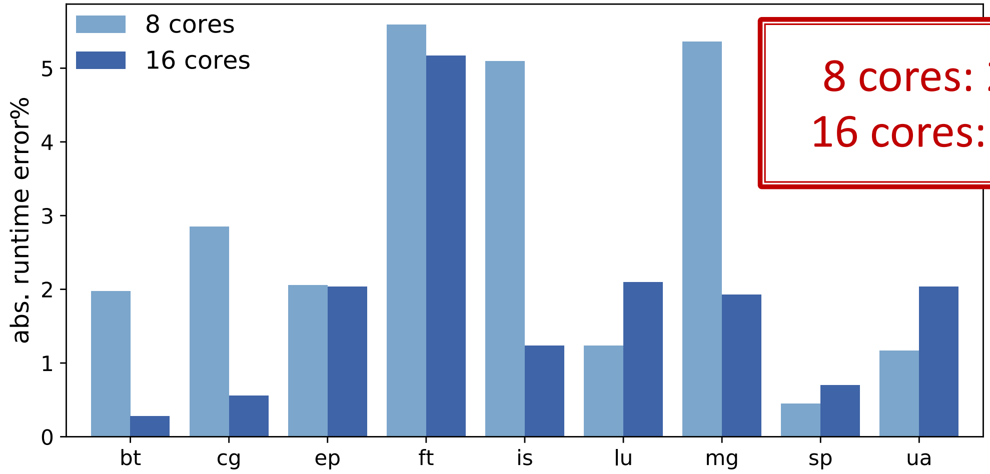




Scaling with Threads

- Runtime prediction error wrt. whole application runtime

NPB 3.3 with *Class C* inputs, *8* and 16 threads, *passive* wait-policy

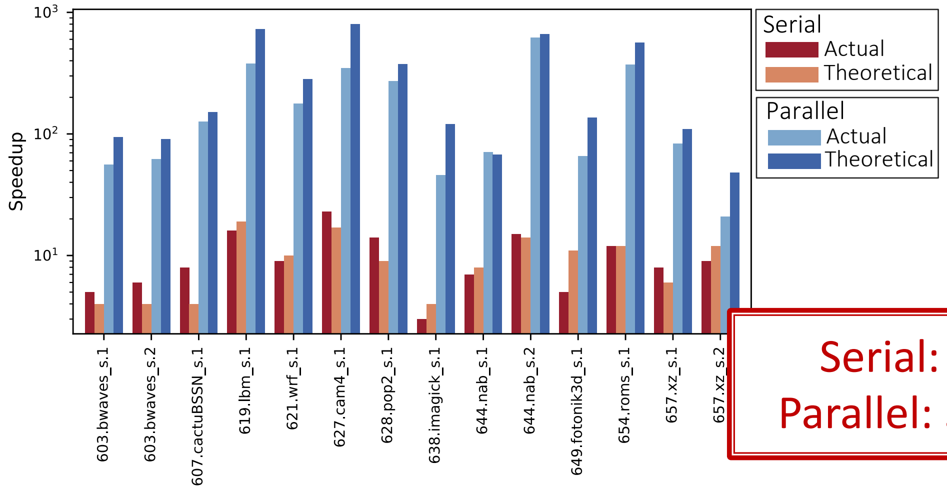


8 cores: 2.87%
16 cores: 1.78%

Speedup

- Parallel and serial speedup achieved for LoopPoint

SPEC CPU2017 with *train* inputs, *8* threads, *active* wait-policy

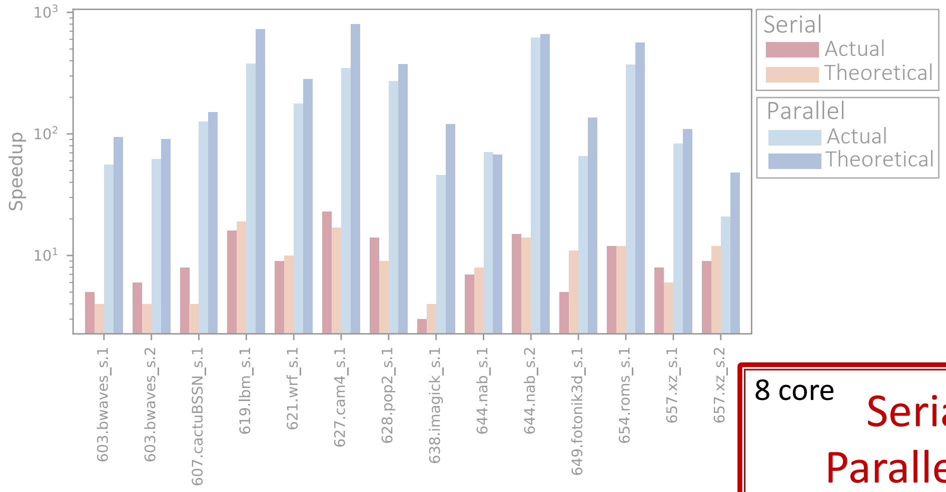


Serial: 9×
Parallel: 303×

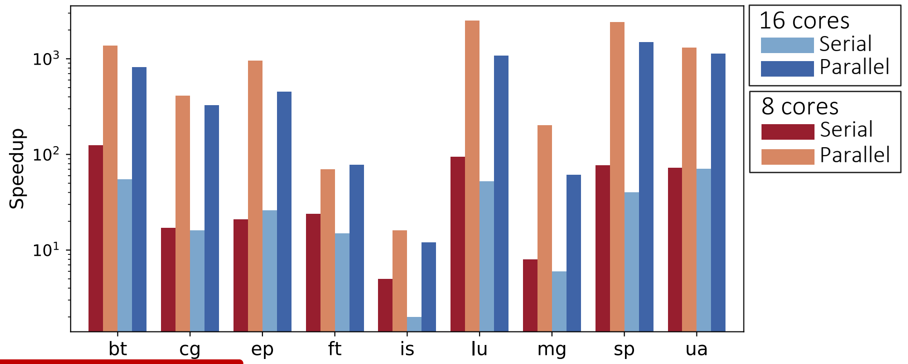
Speedup

- Parallel and serial speedup achieved for LoopPoint

SPEC CPU2017 with *train* inputs, *8* threads, *active* wait-policy



NPB with *Class C* inputs, *8 and 16* threads, *passive* wait-policy



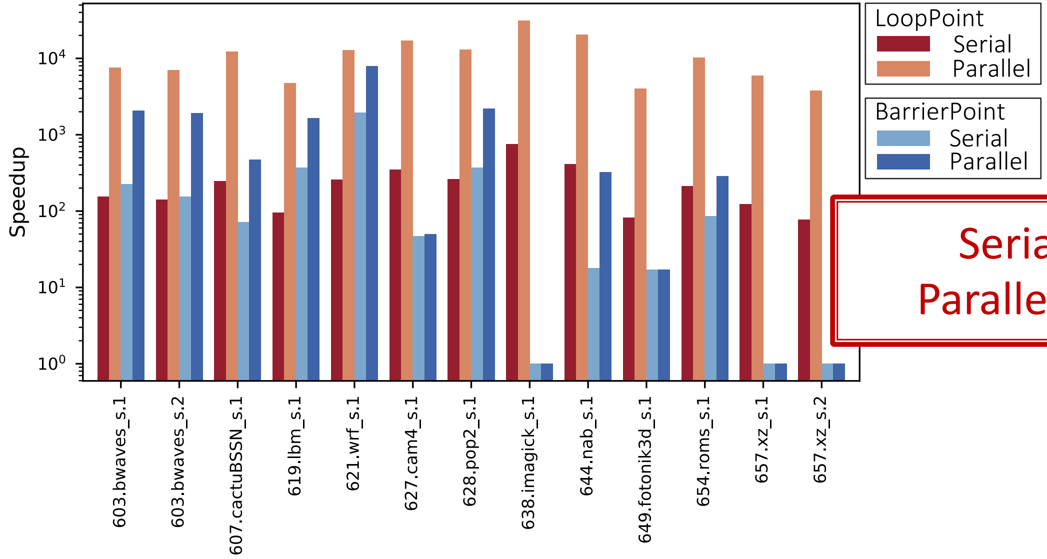
8 core
Serial: 49×
Parallel: 1031×

16 core
Serial: 31×
Parallel: 606×

Speedup

- Speedup comparison with BarrierPoint

SPEC CPU2017 with *ref* inputs, *8* threads, *passive* wait-policy

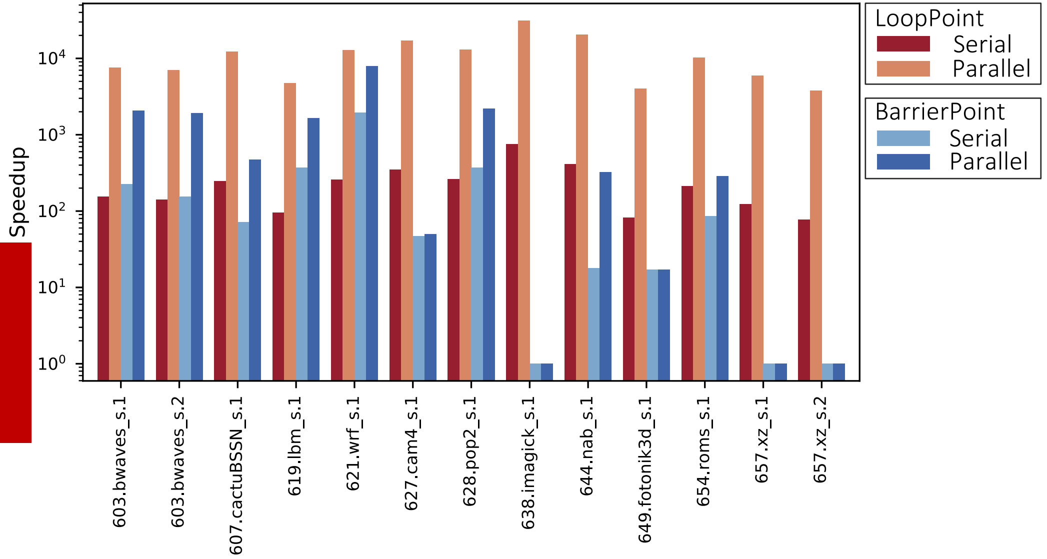


Serial: 244×
Parallel: 11587×

Speedup

- Speedup comparison with BarrierPoint

SPEC CPU2017 with *ref* inputs, *8* threads, *passive* wait-policy



Up to 31000X speedup!

Summary

- Contributions
  - Methodology to sample generic multi-threaded workloads
  - Uses application loops as the unit of work
  - Flexible to be used for checkpoint-based simulation

Summary

- Contributions
  - Methodology to sample generic multi-threaded workloads
  - Uses application loops as the unit of work
  - Flexible to be used for checkpoint-based simulation
- Accurate results in minimal time
  - Average absolute error of 2.3% across applications
  - Parallel speedup going up to 31,000 ×
  - Reduces simulation time from a few years to a few hours

There are two kinds of people in this world:

1. *Those who can extrapolate from incomplete data*

Thank you!

LoopPoint: Checkpoint-driven Sampled Simulation for Multi-threaded Applications

Alen Sabu[1], Harish Patil[2], Wim Heirman[2], Trevor E. Carlson[1]

alens@comp.nus.edu.sg, harish.patil@intel.com, wim.heirman@intel.com, tcarlson@comp.nus.edu.sg

[1]National University of Singapore

[2]Intel Corporation