IISWC 2020 Tutorial: Proxy Benchmarks for Reproducible Research

Lizy K. John, Steven Flolid, Zachary Susskind
UT Austin
&
Emily Shriver (Intel Labs)

This work is supported currently by NSF grant 1745813 and an Intel faculty award. Proxy benchmark research in the past has been supported in the past by IBM, SRC, and AMD.

Outline

- Introduction to Proxy Benchmarks
  - Miniaturization, Software Stack Abstraction, Proprietary Codes
- Metrics, Techniques and Results for early Proxies
  - Bell, Joshi, Ganesan, and Panda Proxies; Results
- SimPoint as an alternative (CPU 2017 SimPoints) – Steven Flolid
- Challenges - Improving Accuracy of Proxy Benchmarks
  - Branch and Memory Behavior – Problem and Current State
  - Presentations by Zachary Susskind and Steven Flolid
- Industry Use
  - Intel – Presentation from Emily Shriver
- ISA Independence - LLVM Proxies
- Reproducibility of Research using Proxies
- Other Applications of Proxy Code Generator

Introduction

- Bad Press on Early Synthetic Benchmarks
- Whetstone, Dhrystone
- Misuse of Synthetic Benchmarks
- Not many good memories
- A totally new look at a new kind of synthetic benchmarks
- A few new scenarios
  - **The Pre-Silicon Nightmare**
  - **Proprietary Applications that cannot be shared**
  - **Expiring Benchmarks**
  - **Power and Thermal Stress Benchmarks**

The Presilicon Design Nightmare

Modern microprocessors are built from millions of lines of VHDL/Verilog code and billions of transistors

RTL and Performance Models are 1000X or more slower than hardware.

Industry standard benchmarks run for hours on actual hardware.

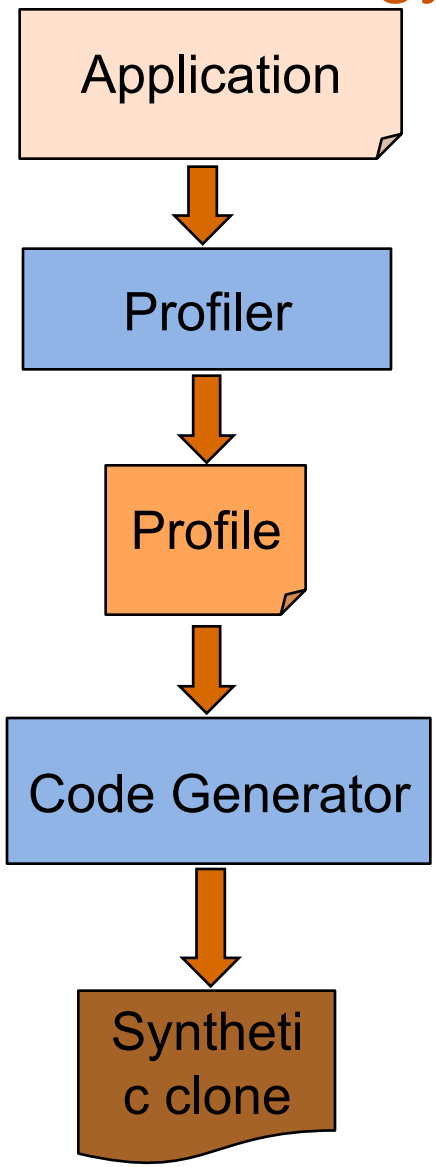
Design tradeoffs need to be analyzed quickly to make timely design decisions.

If the wrong decisions are made, the presilicon nightmare will become a serious

TPC Transaction Processing Performance Council

postsilicon nightmare

Methodology: Workload Modeling and Synthesis



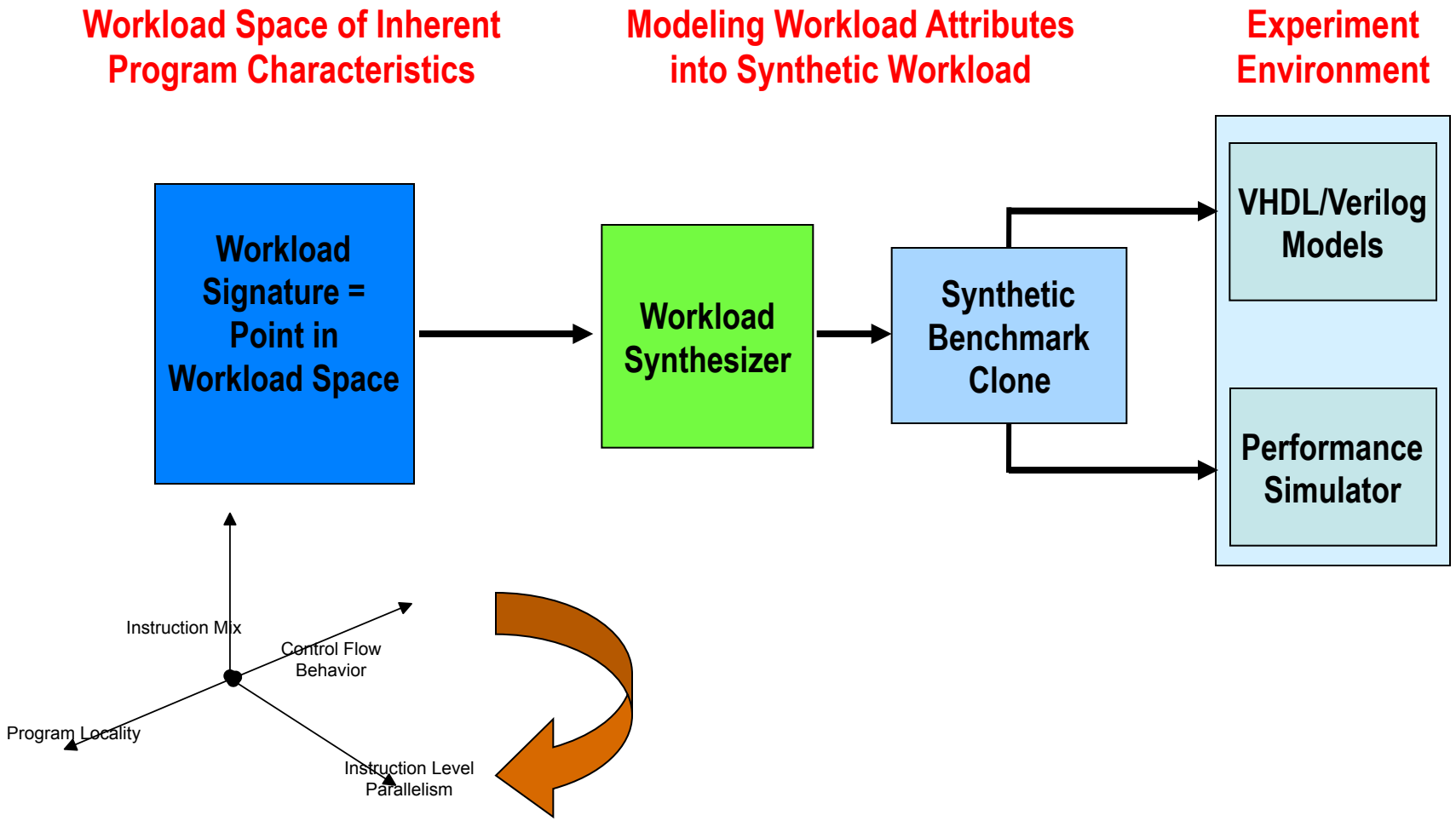
1. **Identify a list of intrinsic properties $\alpha, \beta, \gamma \ldots \omega$ to uniquely describe the behavior of a modern program**

2. **Develop a methodology to reproduce the same behavior synthetically**

Representing a Program with Intrinsic Properties

- **Is it possible to find a list of intrinsic properties α,β,γ, ...ω to uniquely describe a program?**
- **What will be the characteristics α,β,γ, ...ω?**
- **Platform (Micro-architecture) Independent Characteristics**
  - i-mix, locality, address patterns, dependency
- **Platform (Micro-architecture) Dependent Characteristics**
  - Cache-hit rates, IPC/throughput, coherency traffic

Proxy Workload Generation



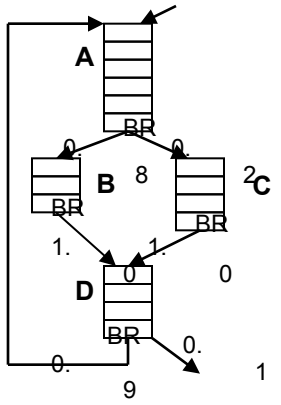
Miniaturized Proxy Generation Process

Real Workload

```
10101010101001010
10101010101010100
10101010100101010
10101010101010010
10101010101001010
10101010010010101
10101010010101100
10101010101001010
10101010101010100
10101010010010101
10101010101010101
10101010101010100
10101010101010110
10101010010101010
10101010101010101
10101010101010110
101010110
```

Detailed Workload Profile

Instruction Mix, ILP, Data Access Patterns, branch patterns ..




```
ADD R1, R2,R3
LD R4, R1, R6
MUL R3, R6, R7
ADD R3, R2, R5
DIV R10, R2, R1
SUB R3, R5, R6
STORE R3, R10, R20
ADD R1, R2,R3
LD R4, R1, R6
MUL R3, R6, R7
ADD R3, R2, R5
DIV R10, R2, R1
SUB R3, R5, R1
BEQ R3, R6, LOOP
SUB R3, R5, R6
STORE R3, R10, R20
DIV R10, R2, R1
.....
```

Synthetic Clone

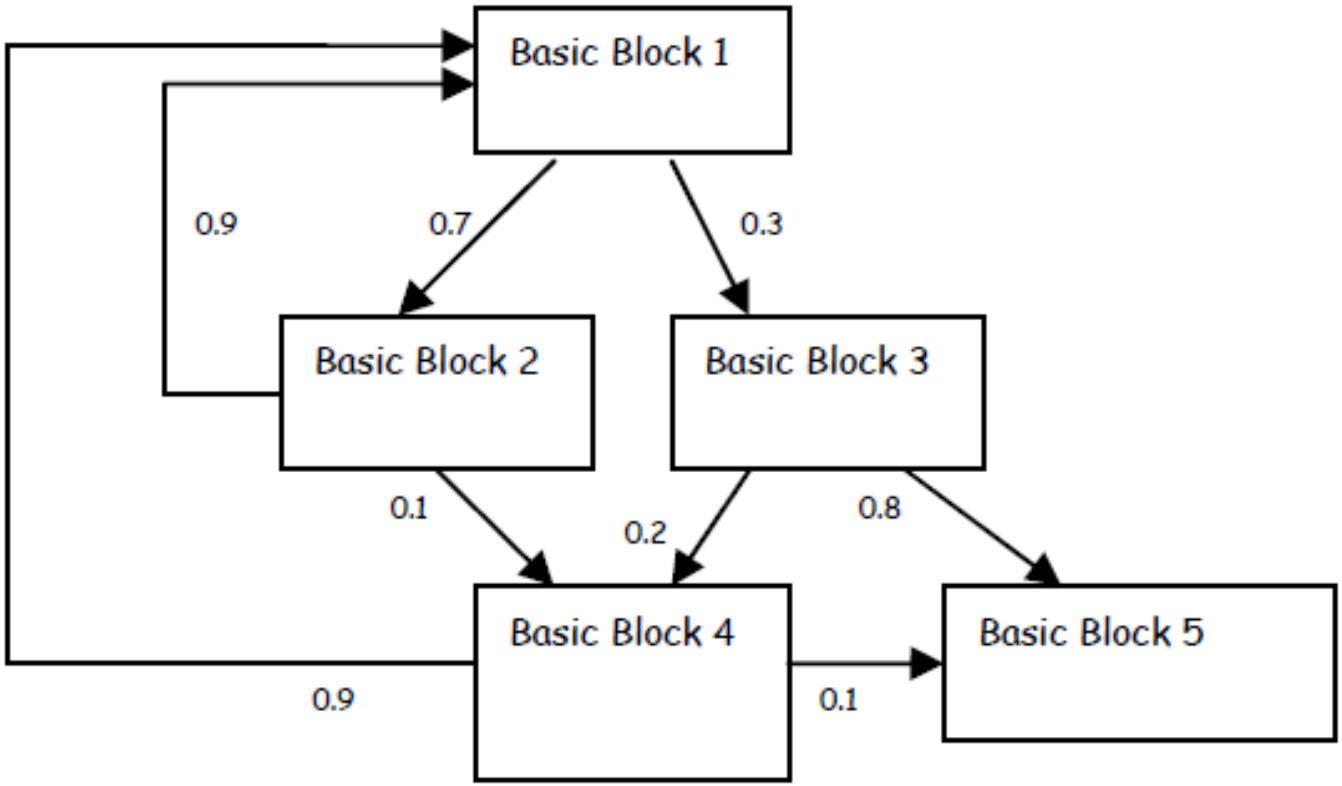
Measure Workload Characteristics at Basic Block Granularity

Generate Synthetic Clone with Similar Inherent Characteristics

Steps in Proxy generation

- Generate Workload profile
  - Command:./mkJSON_both.sh $input $output $output_name.fixed
- Generate Proxy
  - Command: ./CodeGenerator.py --configFile=$file –o $output
- Run Proxy
  - Command: c executable

Statistical Flow Graph (SFG)



Profiling the Original Workload

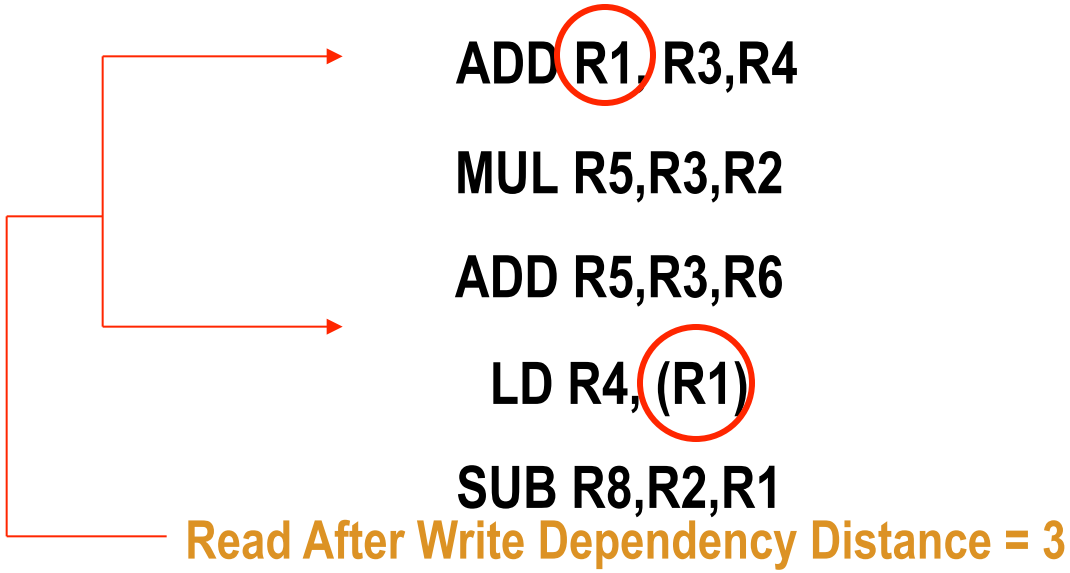
No.	Metric	Category
1	Dynamic execution freq. of basic blocks	Control flow predictability
2	Average basic block size	
3	Branch taken rate for each branch	
4	Instruction pattern in a basic block	
5	Branch transition rate for each branch present in the workload	
6	% Integer multiplication	Instruction mix
7	% Integer division	
8	% FP multiplication	
9	%FP division	
10	% Loads	
11	% Stores	
12	% Branches	
13	Dependency distance distribution per instruction type	Instruction level parallelism
14	Stride value per static load/store	Data locality
15	Data Footprint of the workload	
16	Mean and standard deviation of the MLP	Memory level parallelism
17	Mean number of consecutive dynamic inst. when no outstanding long-latency loads	

Abstract Workload Model

No.	Metric	Category
1	Basic block size	Control flow predictability
2	Branch taken rate for each branch	
3	Branch transition rate	
4	Proportion of INT ALU, INT MUL, INT DIV, FP ADD, FP MUL, FP DIV, FP MOV, FP SQRT, LOAD & STORE	Instruction mix
5	Dependency distance distribution	Instruction level parallelism
6	Private stride value per static load/store	Data locality
7	Data Footprint of the workload	
8	Mean and standard deviation of the MLP	Memory Level Parallelism (MLP)
9	MLP frequency	
10	Number of threads	Thread level parallelism
11	**Thread class and processor assignment**	**Shared data access pattern and communication characteristics**
12	**Percentage loads to private data**	
13	**Percentage loads to read-only data**	
14	**Percentage migratory loads**	
15	**Percentage consumer loads**	
16	**Percentage irregular loads**	
17	**Percentage stores to private data**	
18	**Percentage producer stores**	
19	**Percentage irregular stores**	
20	**Shared stride value per static load/store**	
21	**Data pool distribution based on sharing patterns**	
22	Number of lock/unlock pairs and	Synchronization Characteristics
23	Number of mutex objects	
24	Number of Instructions between lock and unlock	

Instruction Dependency Distance

- **Instruction level parallelism:**
  - Lower dependency distance corresponds to lower ILP and vice versa.



CFG Information - SPEC CPU2006

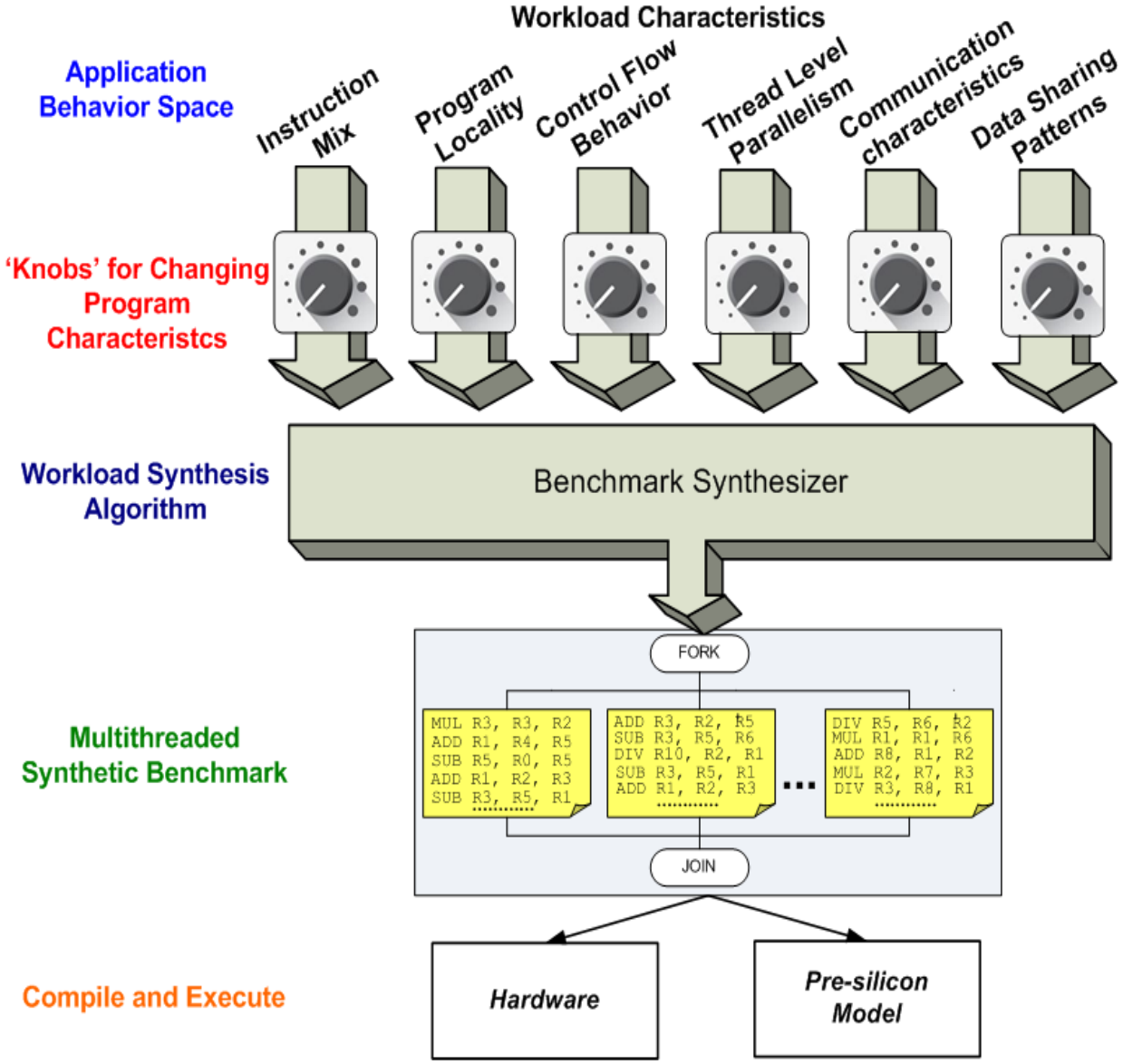
Benchmark	No. of B.Blks	# B.Blks for 90% of Prog Exec.	Branch Transition Rate	Average B.Blk Size	Average Number of Succ. B.Blks
CINT 2006					
400.perlbench	**1620**	**169**	0.07	5.99	1.93
401.bzip2	93	15	0.06	7.91	1.94
445.gobmk	617	129	0.15	6.97	2.42
456.hmmer	142	12	0.10	8.12	1.49
458.sjeng	281	70	0.17	5.97	2.54
462.libquantum	29	3	0.03	4.59	1.41
464.h264ref	**1074**	**55**	0.09	15.13	1.69
471.omnetpp	443	112	0.08	5.25	2.06
473.astar	96	16	0.19	8.77	1.43
483.xalancbmk	62	11	0.00	3.54	2.38
429.mcf	179	96	0.01	10.58	1.21
403.gcc	698	363	0.08	7.14	1.76
CFP 2006					
410.bwaves	32	10	0.26	32.61	1.50
433.milc	42	18	0.44	15.32	1.48
434.zeusmp	48	13	0.02	43.29	1.67
435.gromacs	6	3	0.07	518.06	1.33
436.cactusADM	22	7	0.18	247.16	1.27
437.leslie3d	**533**	**13**	0.02	20.43	1.74
444.namd	70	8	0.03	18.95	1.61
450.soplex	358	14	0.03	7.79	1.45
459.GemsFDTD	119	10	0.01	48.81	1.40
482.sphinx3	544	20	0.07	12.76	1.70

CFG Information – Implant Bench

Benchmark	No. of B.Blks	# B.Blks for 90% of Prog. Exec.	Branch Transition Rate	Average B.Blk Size	Average # of Succ. B.Blks
AI_Adaline	**342**	**6**	0.05	39.22	1.63
AI_BPN	**410**	**13**	0.06	54.27	1.80
AI_GA	503	160	0.23	7.7	2.26
Bioinformatics_ELO	194	15	0.05	9.42	1.72
Bioinformatics_LMGC	498	111	0.25	7.55	1.91
Genomics_HMM	463	119	0.28	8.03	1.80
Genomics_NJ	196	18	0.17	9.41	2.03
HeartActivity_pNNx	479	97	0.17	6.41	1.68
Physiology_AFVP	679	105	0.26	7.07	1.77
Physiology_ECGSYN	647	142	0.18	8.99	1.65
Reliability_alder32	143	101	0.43	6.81	1.88
Reliability_crc	172	89	0.36	6.1	2.20
Reliability_luhn	155	90	0.40	6.46	2.10
Reliability_reed_solm	255	4	0.02	4.71	1.93
Security_haval	338	82	0.25	8.1	1.94
Security_KHAZAD	198	17	0.14	121.65	1.97
Security_sha2	313	7	0.06	37.32	1.76

- **Dynamic number of instructions varies from hundreds million to few thousands**

Workload Synthesis Framework

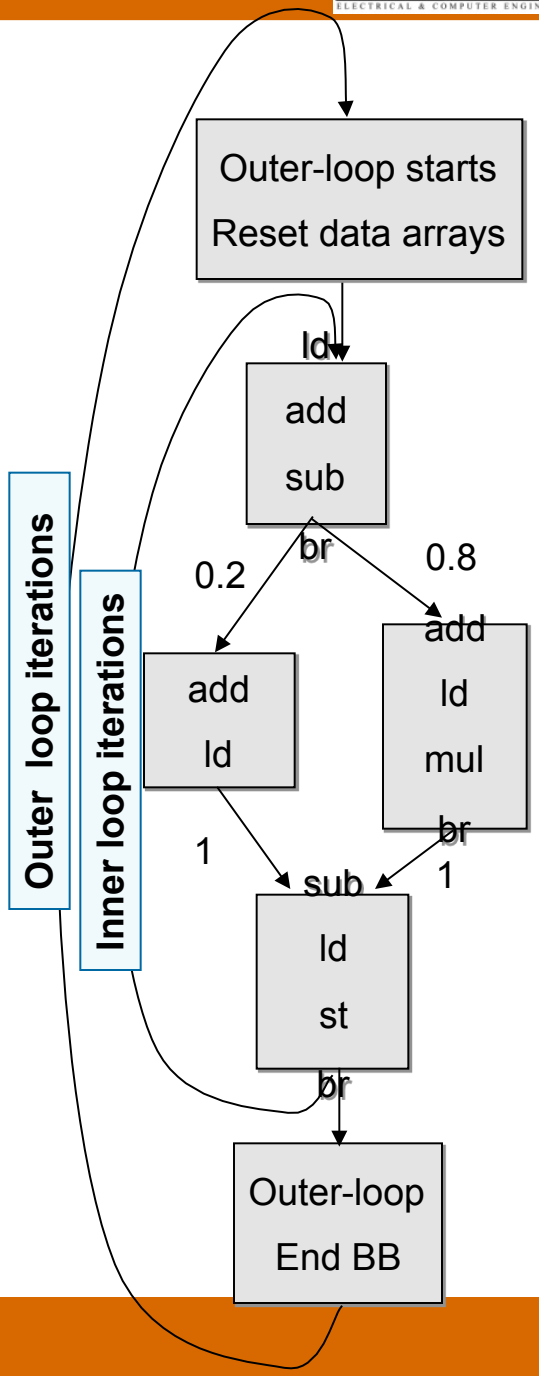


Workload Synthesis Algorithm

1. Generate program spine using basic block information.
2. Populate basic block with instructions satisfying target IMIX
   - Model inter-instruction dependencies
     - Assign producer-consumer chains
4. Memory instruction stride/offset assignment
   - Stride-based memory model
5. Model branch predictability
   - Insert test operation before every branch instruction to control branch outcome
6. Code generation
   - GCC inline assembly for x86

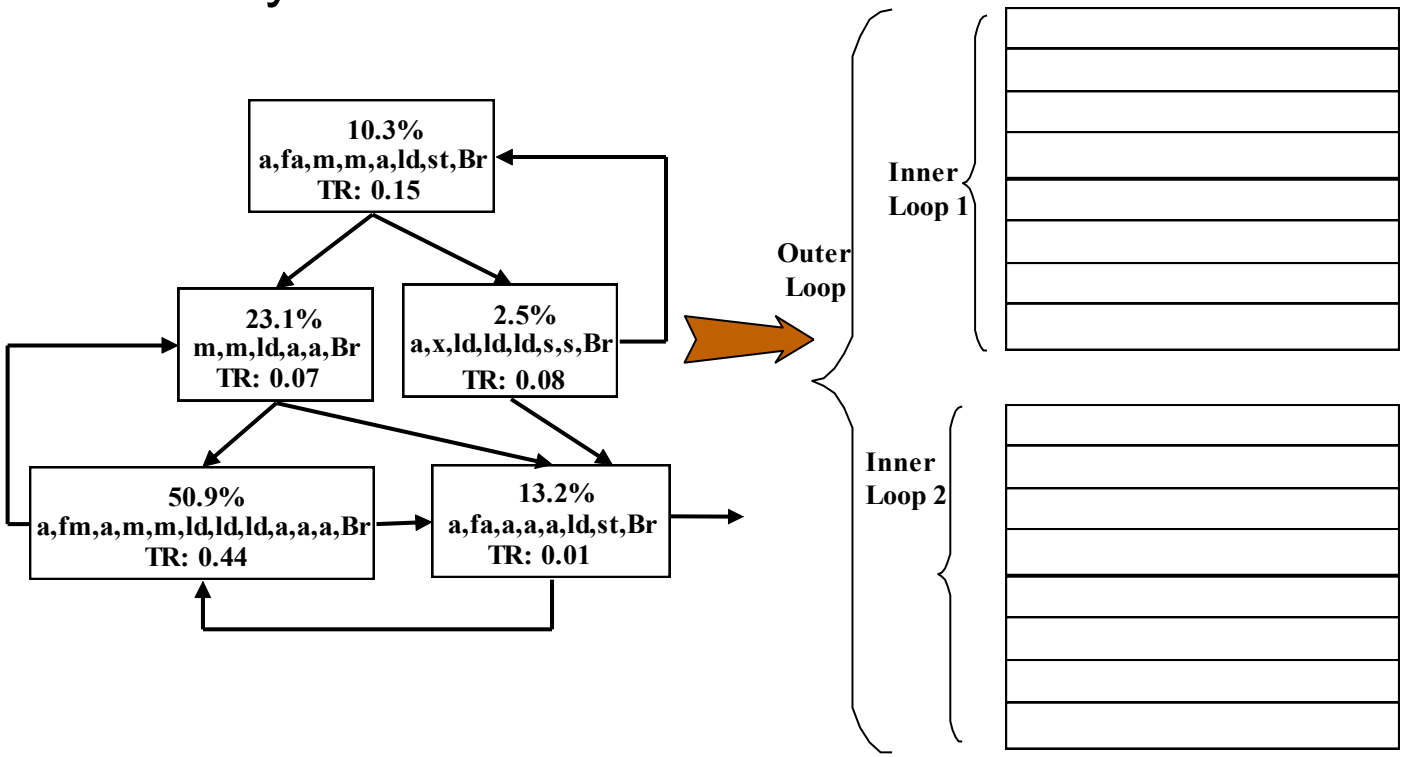
C code with embedded assembly instructions

- Outer loop controls the # of dynamic instructions
- Inner loop iterations control the data footprint



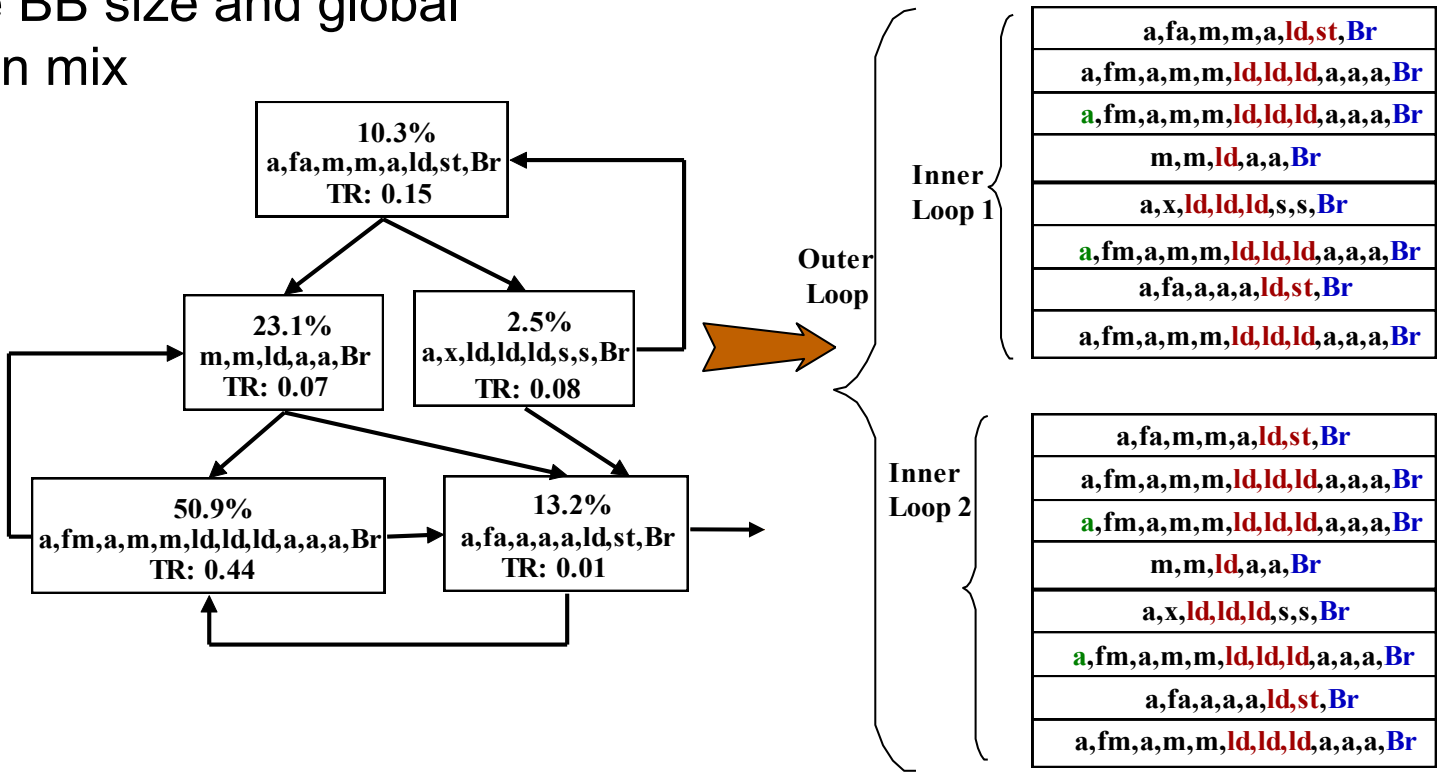
Code Generation

- **Step 1:**
  - Usage of two loops
  - Based on Instruction footprint of original, fix the number of basic blocks in the synthetic



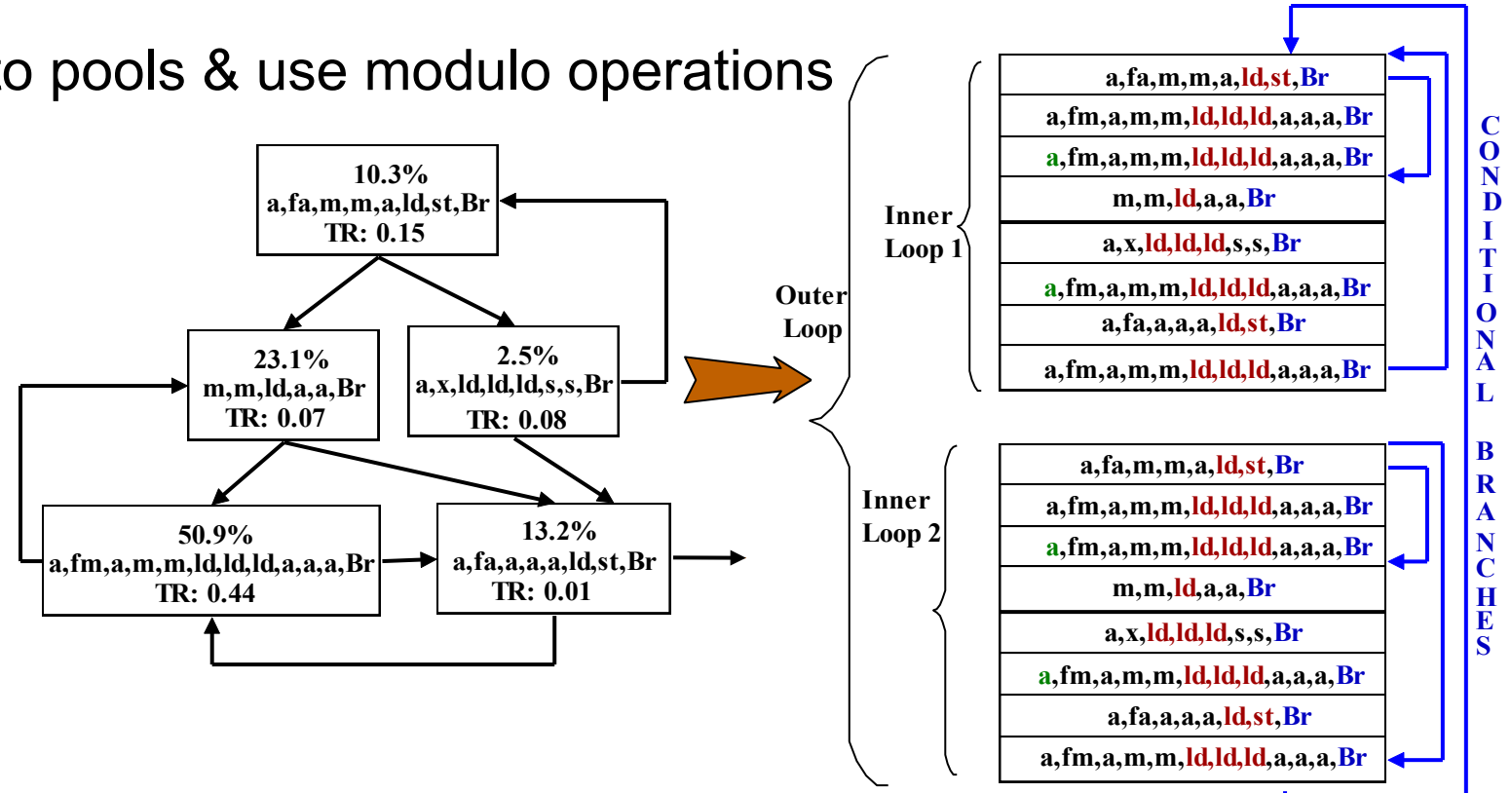
Code Generation

- **Step 2: For each Basic Block**
  - Choose the instruction pattern from a pool of instruction patterns based on the frequency in the profile in terms of Instruction type
  - If not use BB size and global Instruction mix



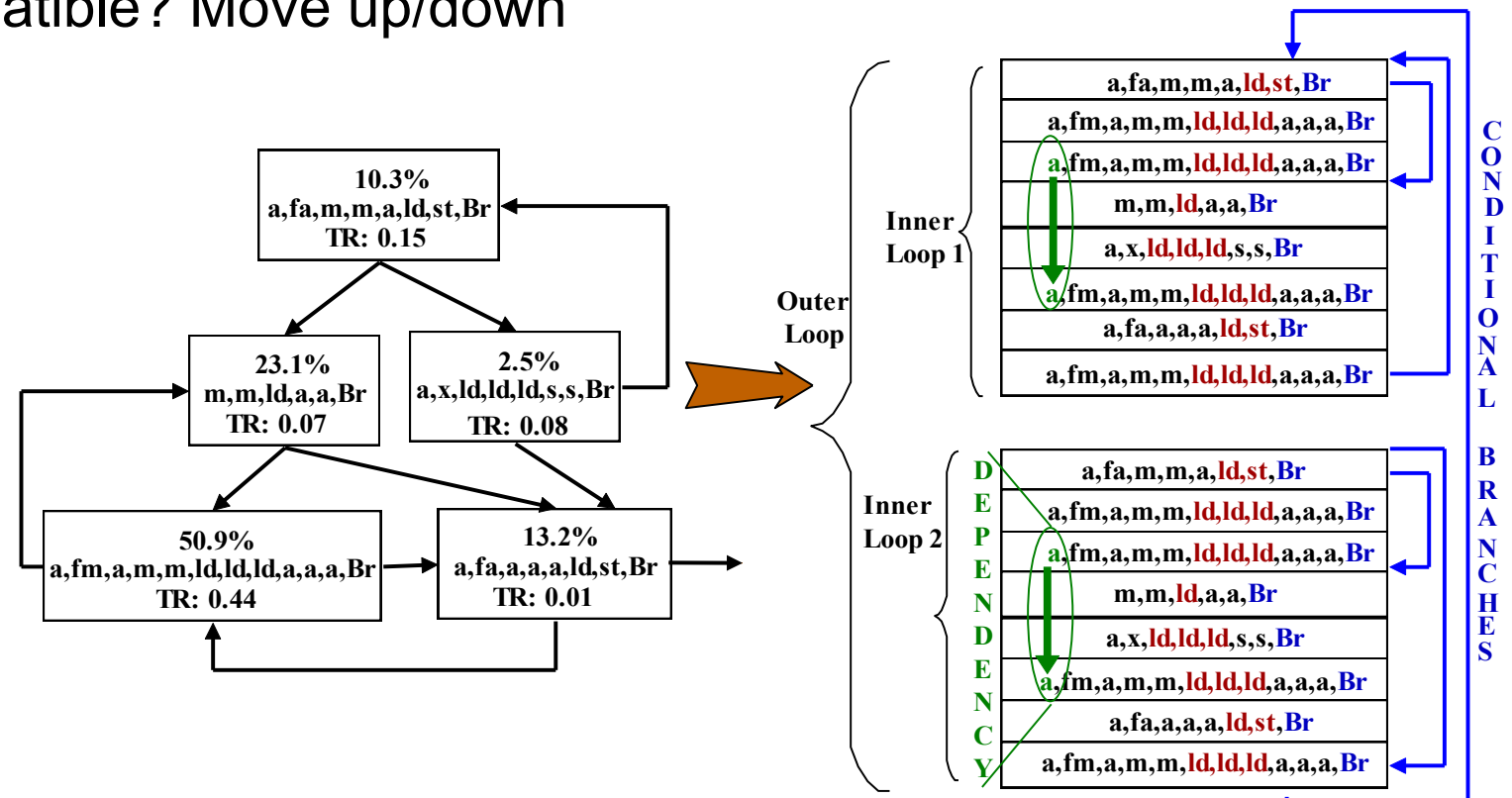
Code Generation

- **Step 3: Bind the basic blocks together using conditional jumps**
  - Low Transition Rate ->always taken/not taken
  - Group into pools & use modulo operations



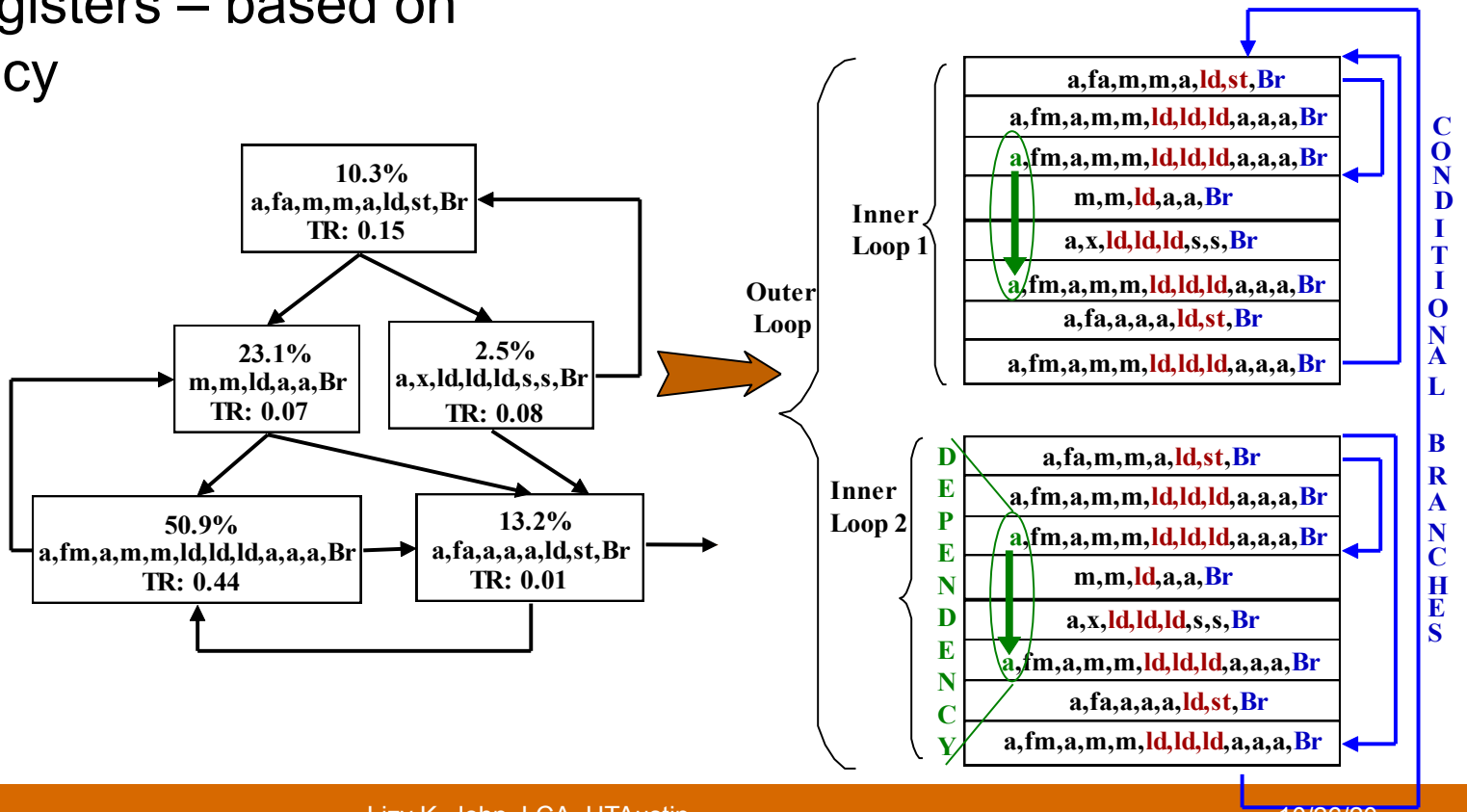
Code Generation

- **Step 4: For each instruction**
  - Find a producer instn to assign a register dependency
  - Not compatible? Move up/down



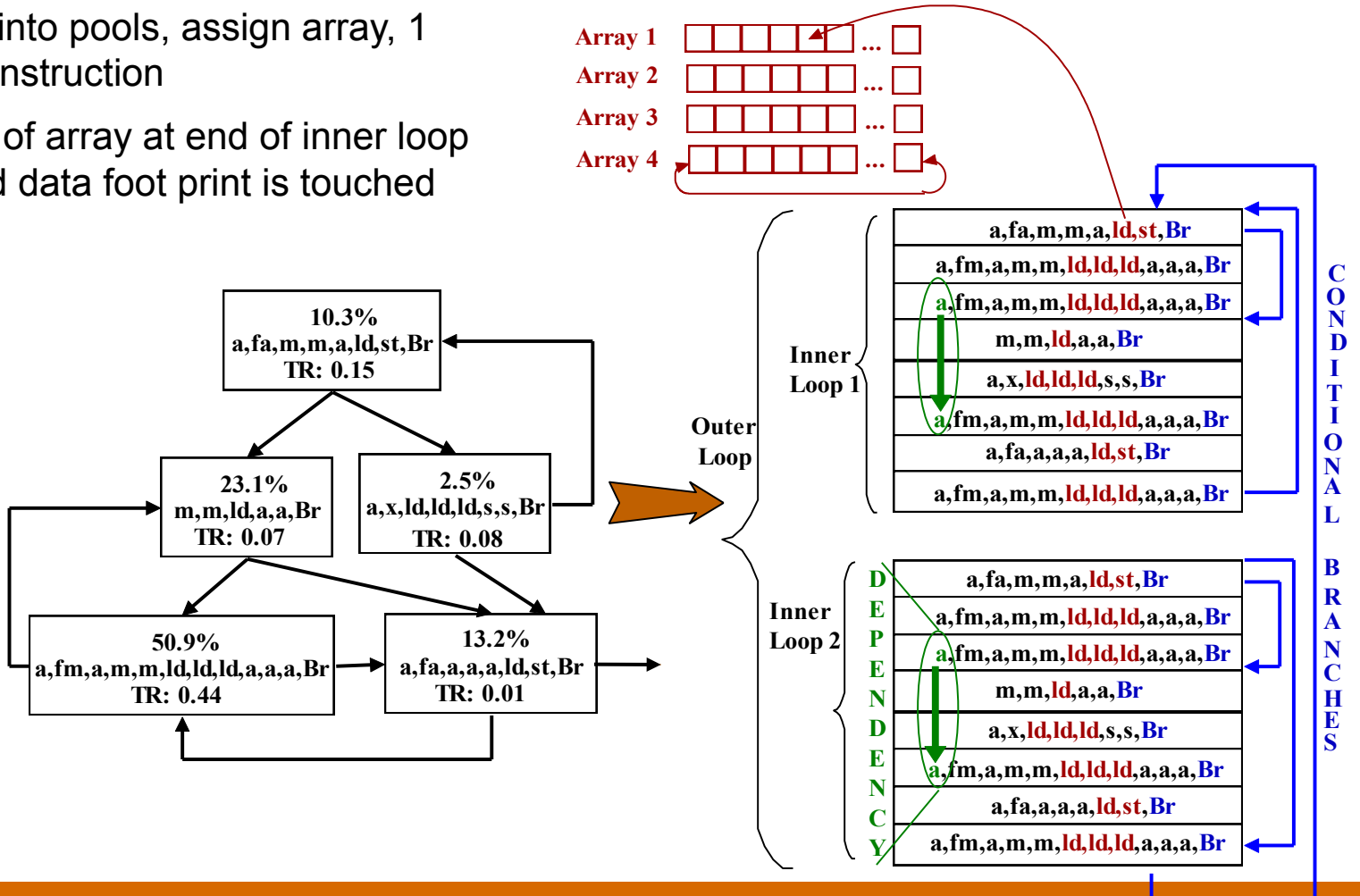
Code Generation

- **Step 5: Assign registers**
  - Destination registers – RoundRobin
  - Source registers – based on dependency



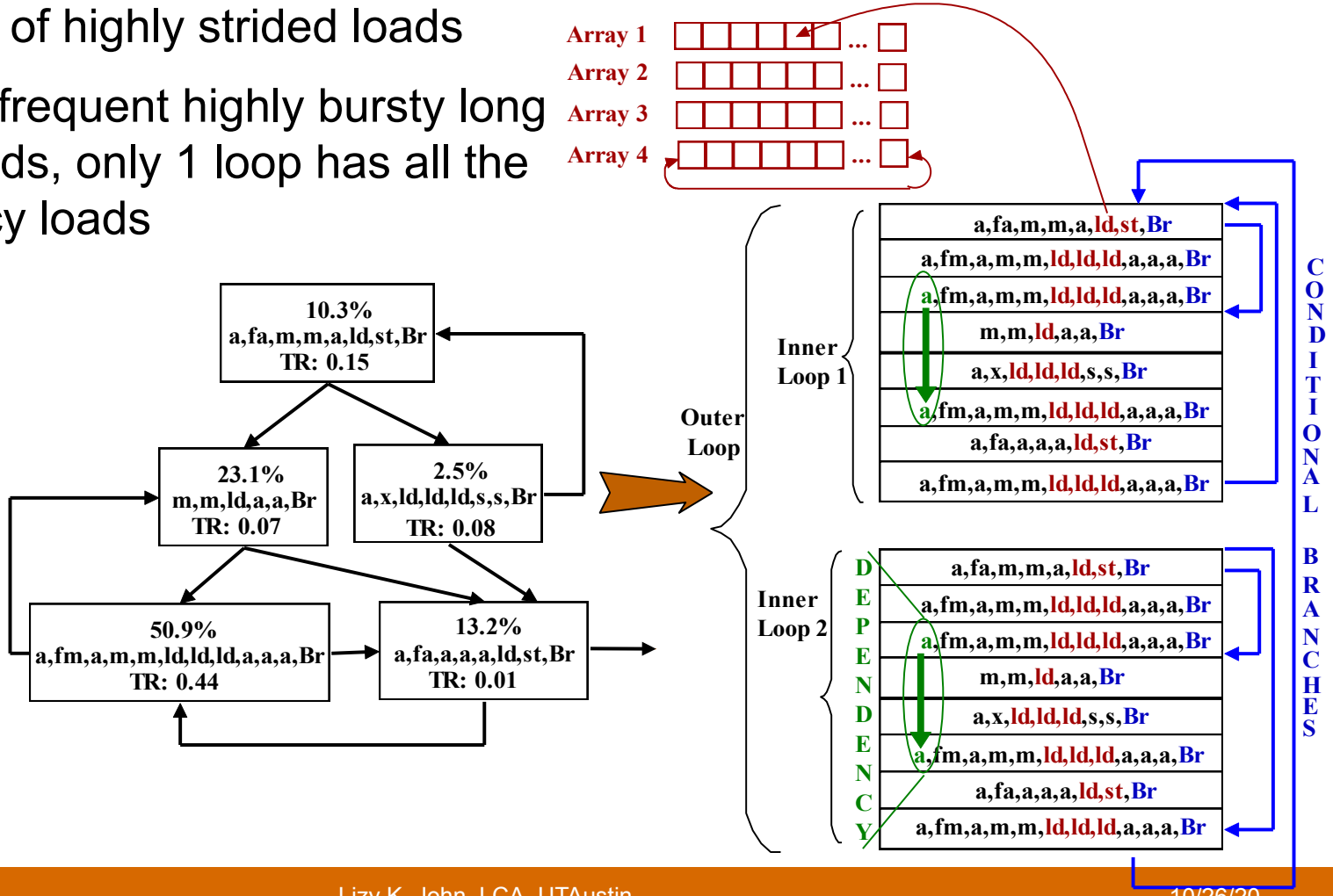
Code Generation

- **Step 6: Memory access model**
  - Ld/st access a set of 1-D arrays in a strided fashion
  - Ld/St - group into pools, assign array, 1 address calc instruction
  - Pointers - top of array at end of inner loop when required data foot print is touched



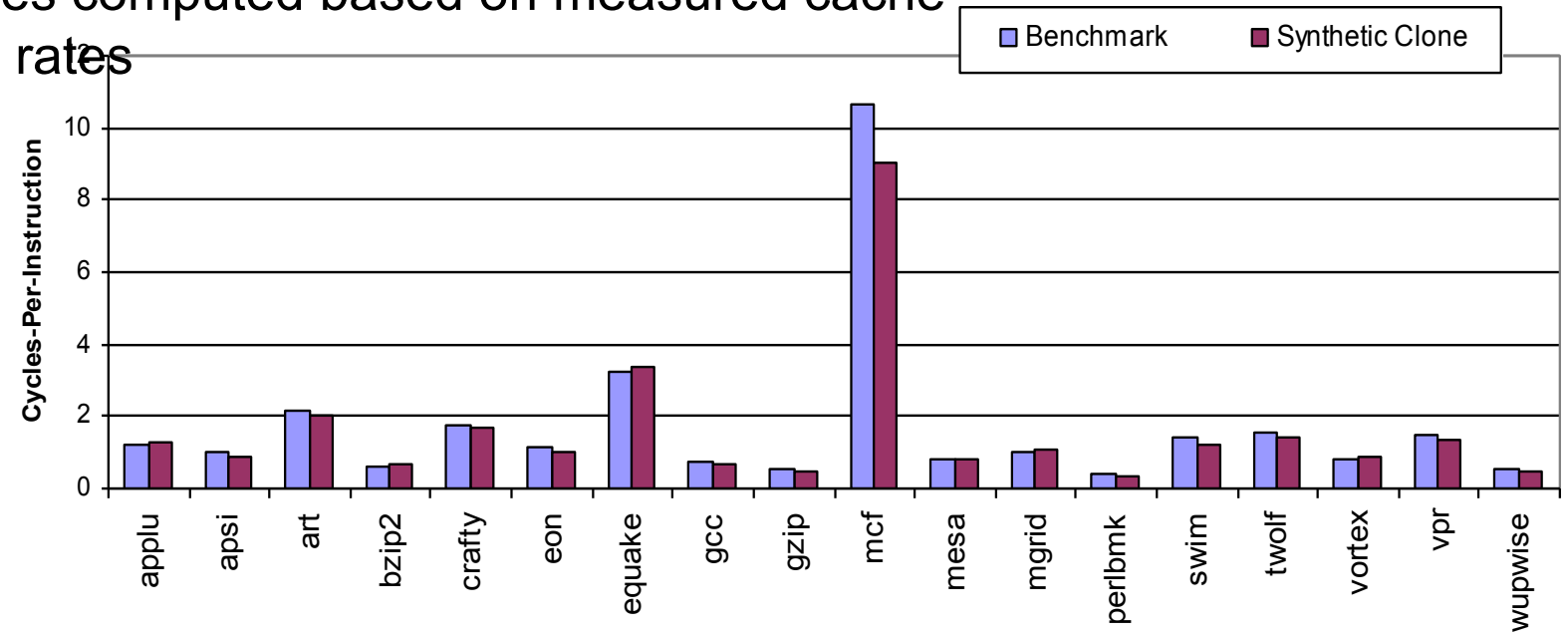
Code Generation

- **Step 6: MLP model**
  - Load-Load dependencies
  - Placement of highly strided loads
  - For very infrequent highly bursty long latency loads, only 1 loop has all the long latency loads



Performance Results from Early Proxies

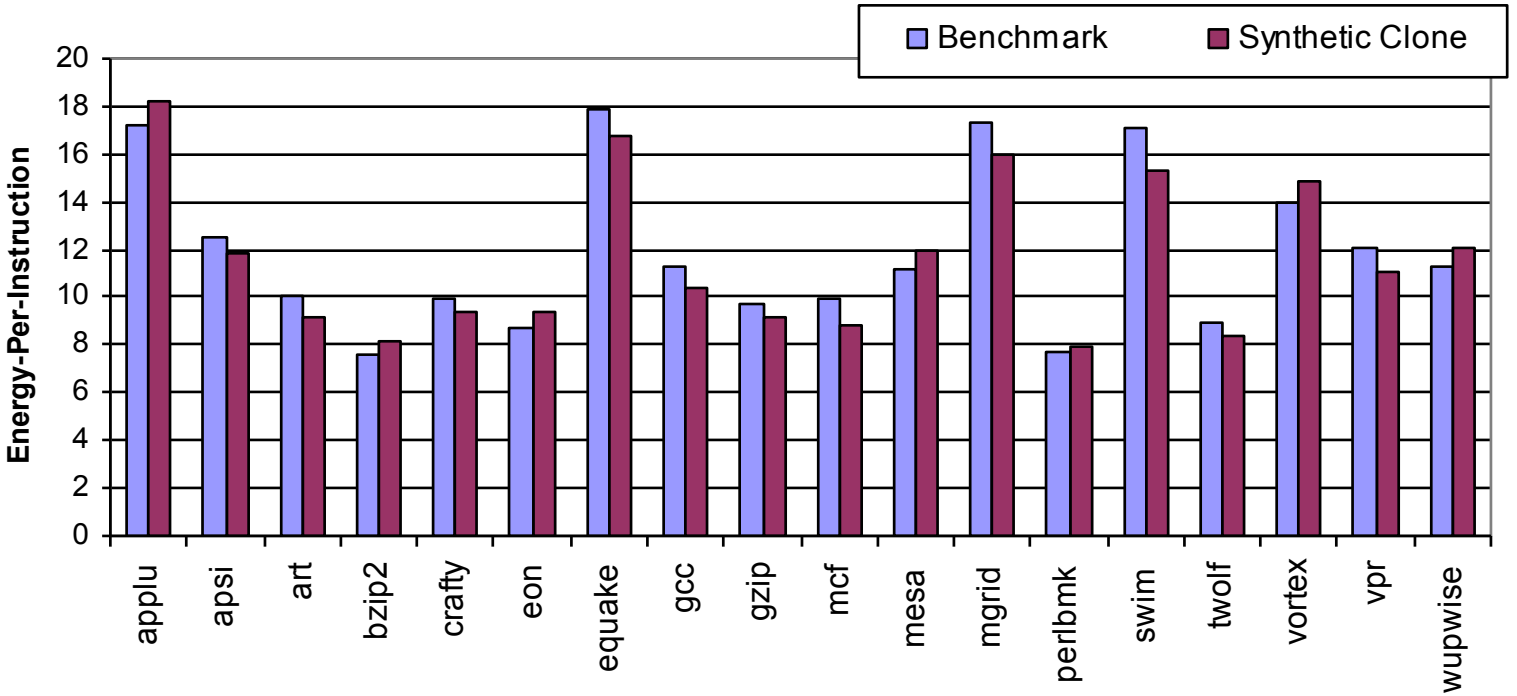
Bell: Some microarch dependent metrics;
Strides computed based on measured cache miss rates



Average Error of 6.3% & Maximum Error of 9.9% (mcf) [Bell05,06]

Overall Power Estimation

Power Results from Bell Proxies



Average Error of 7.3% & Maximum Error of 10.6% (mcf)

Five Orders of Magnitude Speedup!

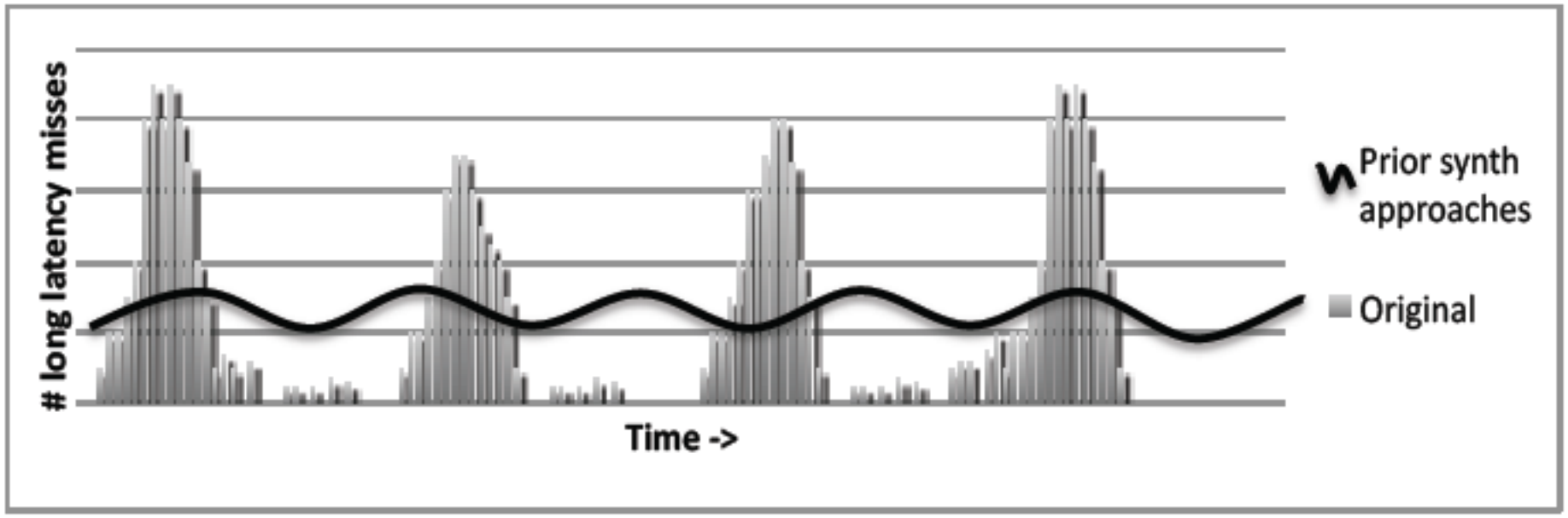
Benchmark	Speedup from Synthetic Benchmark Clone
Applu	22,300
apsi	34,700
art	4,500
bzip2	12,800
crafty	19,100
eon	8,000
equake	13,100
gcc	4,600
gzip	10, 300
mcf	6,100
mesa	14,100
mgrid	41,900
swim	22,500
twolf	34,600
vortex	11,800
vpr	8,400
wupwise	34,900

Improvements from Joshi and Ganesan

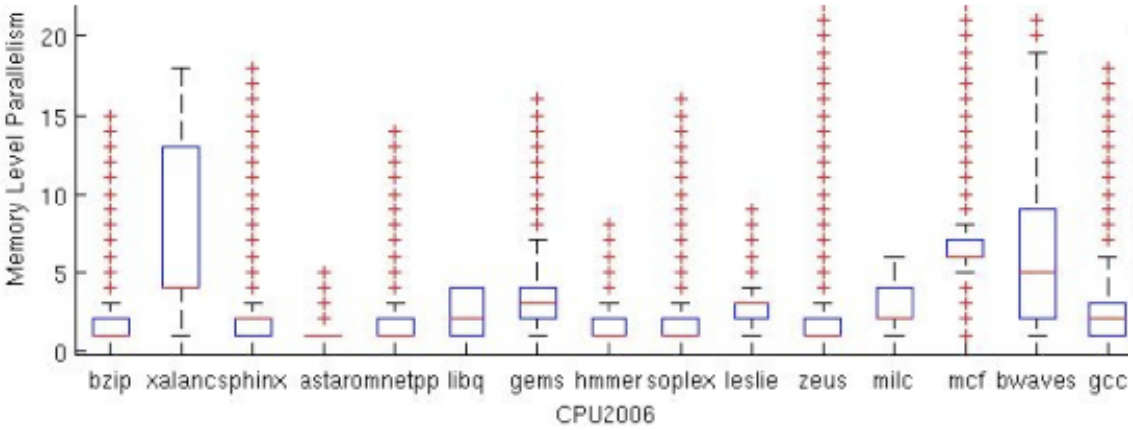
- Joshi added the following metrics:
  - Branch Transition Rate
  - Data Reuse Distance
- Ganesan added the following metrics:
  - MLP (Memory Level Parallelism)
  - Parallel Code Data sharing and Synchronization Metrics

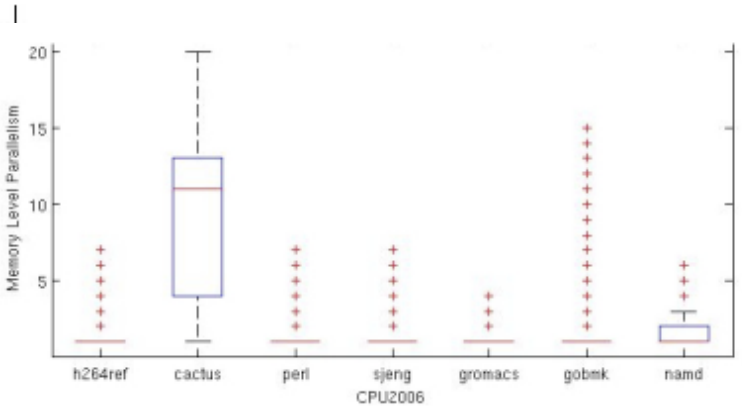
Previous approaches and Memory Level Parallelism

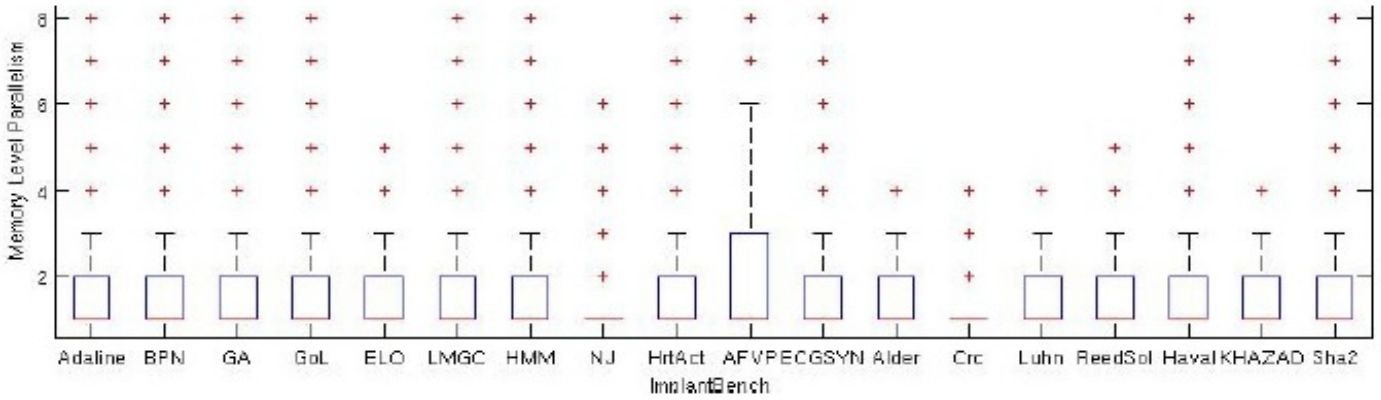
- **Existing synthetic workload generation methodologies Joshi et al [IISWC 06], Bell et al [ISPASS '06, ICS '05]**
  - **Capture control flow, cache access and ILP behavior**
  - **Do not capture the burstiness of long-latency loads or the Memory Level Parallelism (MLP)**
  - **May still match the missrate of the original, but not the performance**



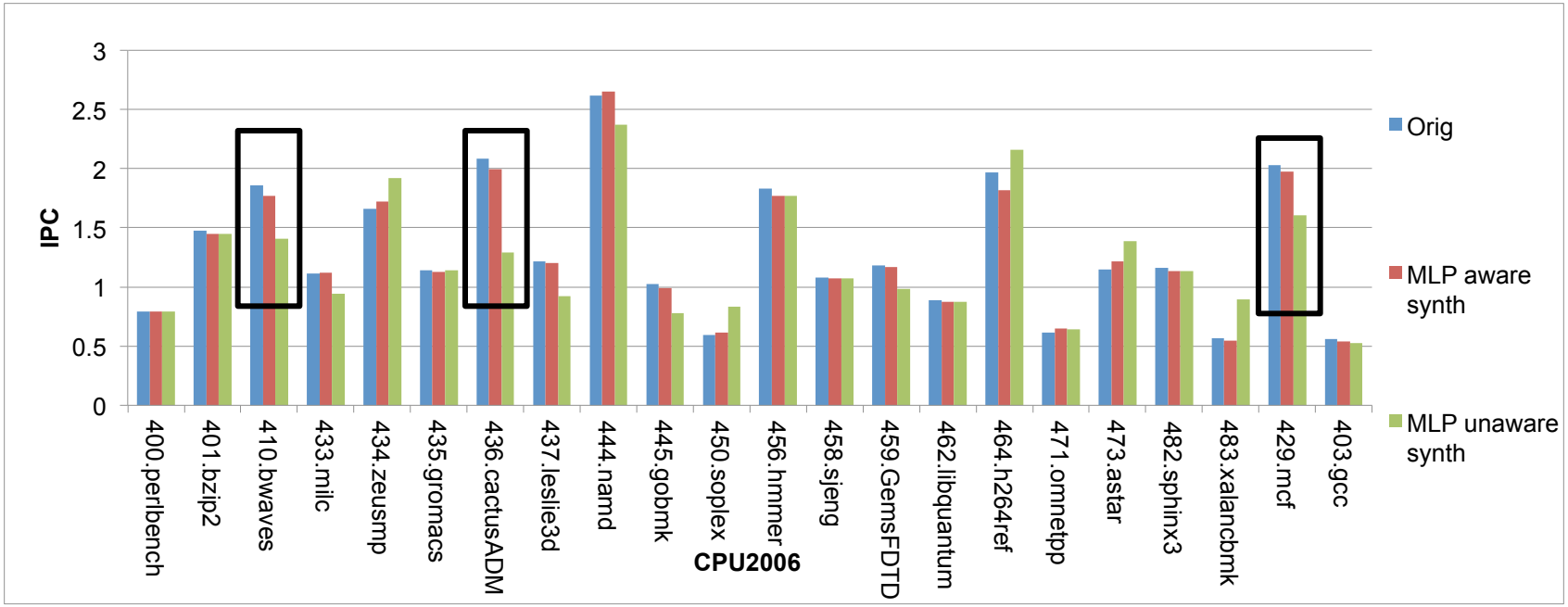
Memory Level Parallelism





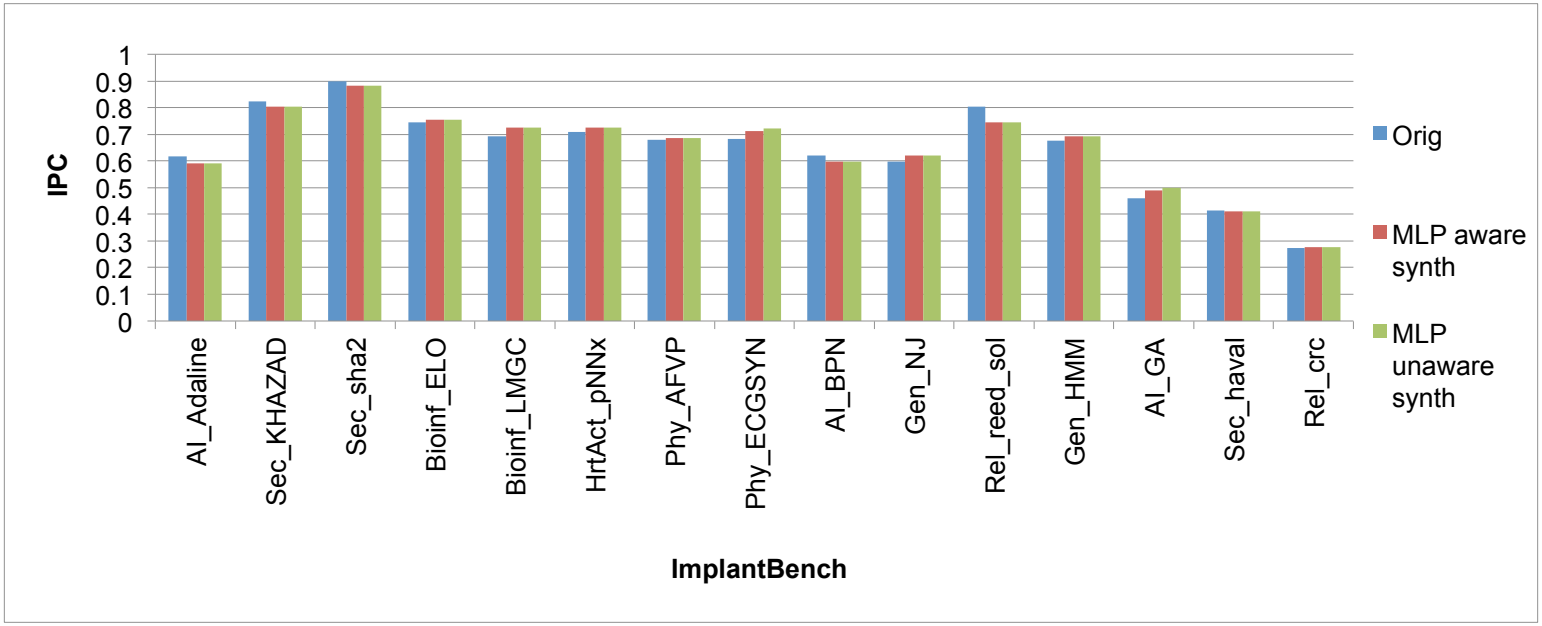


IPC Comparison – SPEC CPU2006



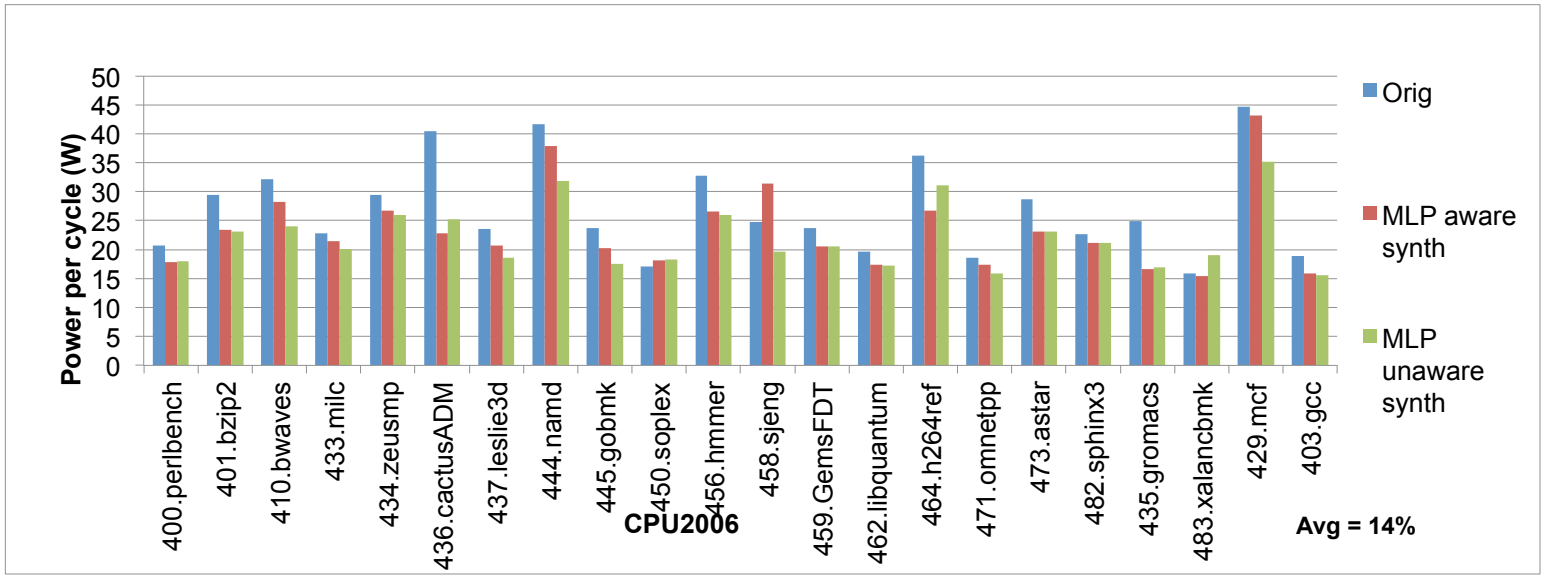
- MLP Aware - Avg = 2.8%, Max = 7.7% for 464.h264ref
- MLP Unaware – Avg = 15.3%

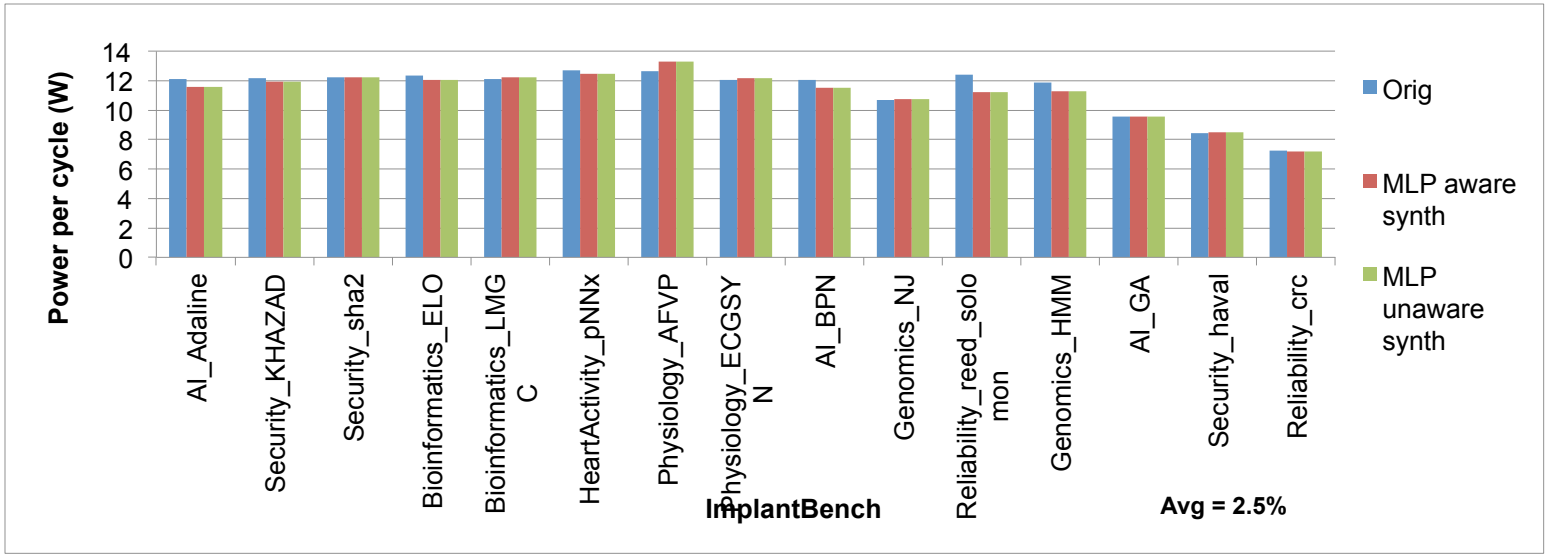
IPC Comparison – ImplantBench



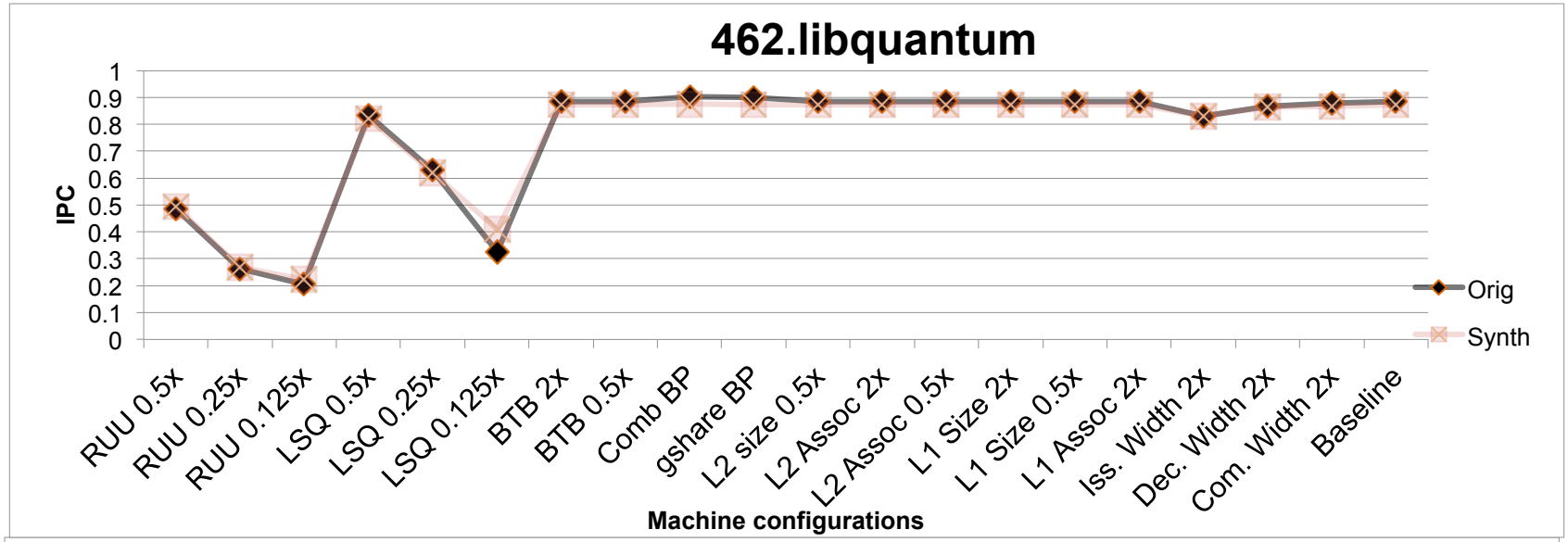
- Avg = 2.9%, Max = 7.2%

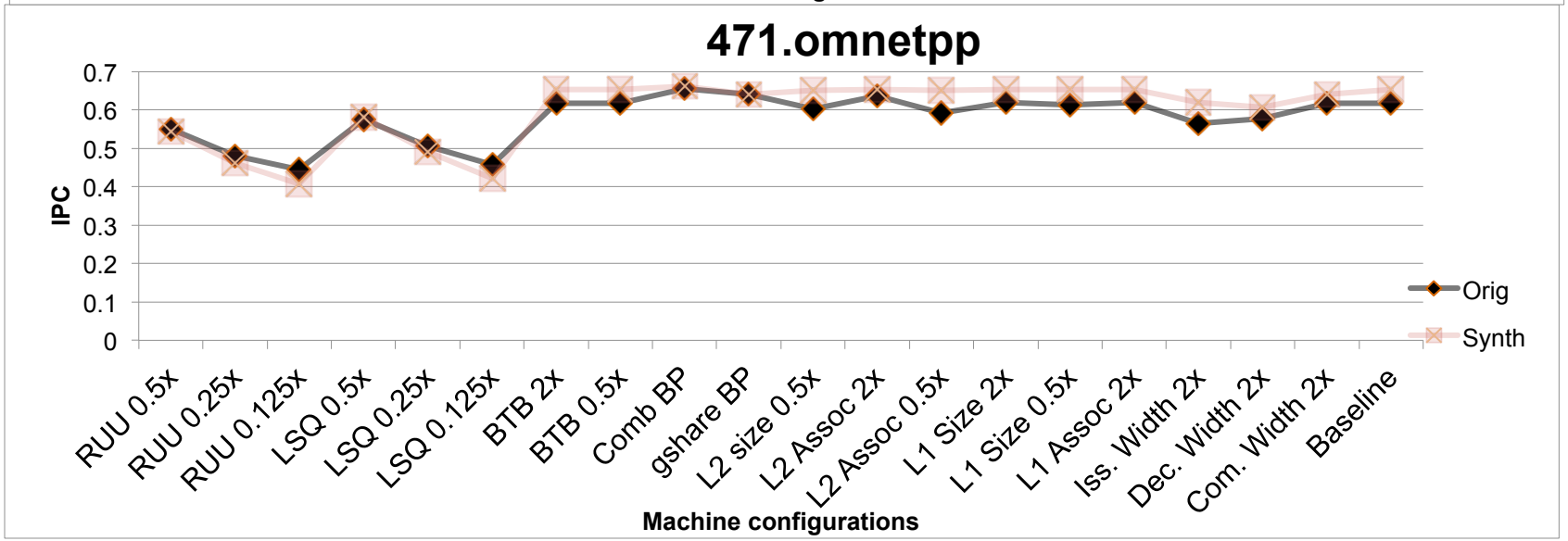
Power Comparison



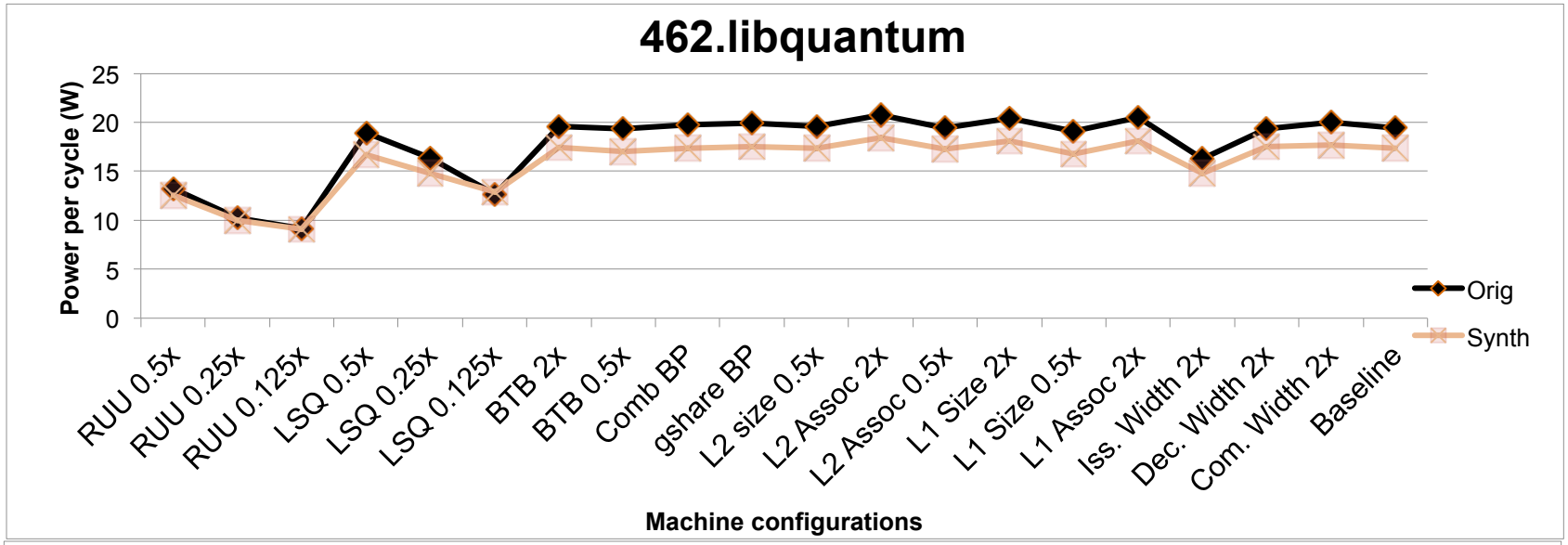


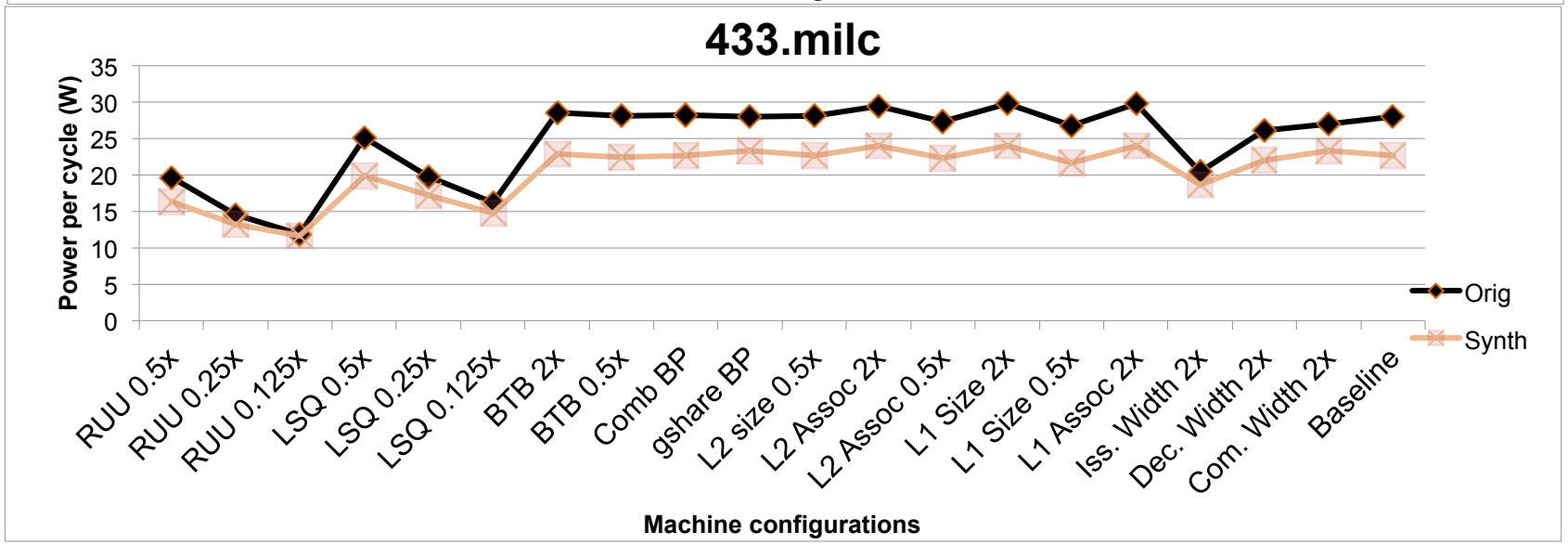
Relative IPC Accuracy for CINT2006 Benchmarks



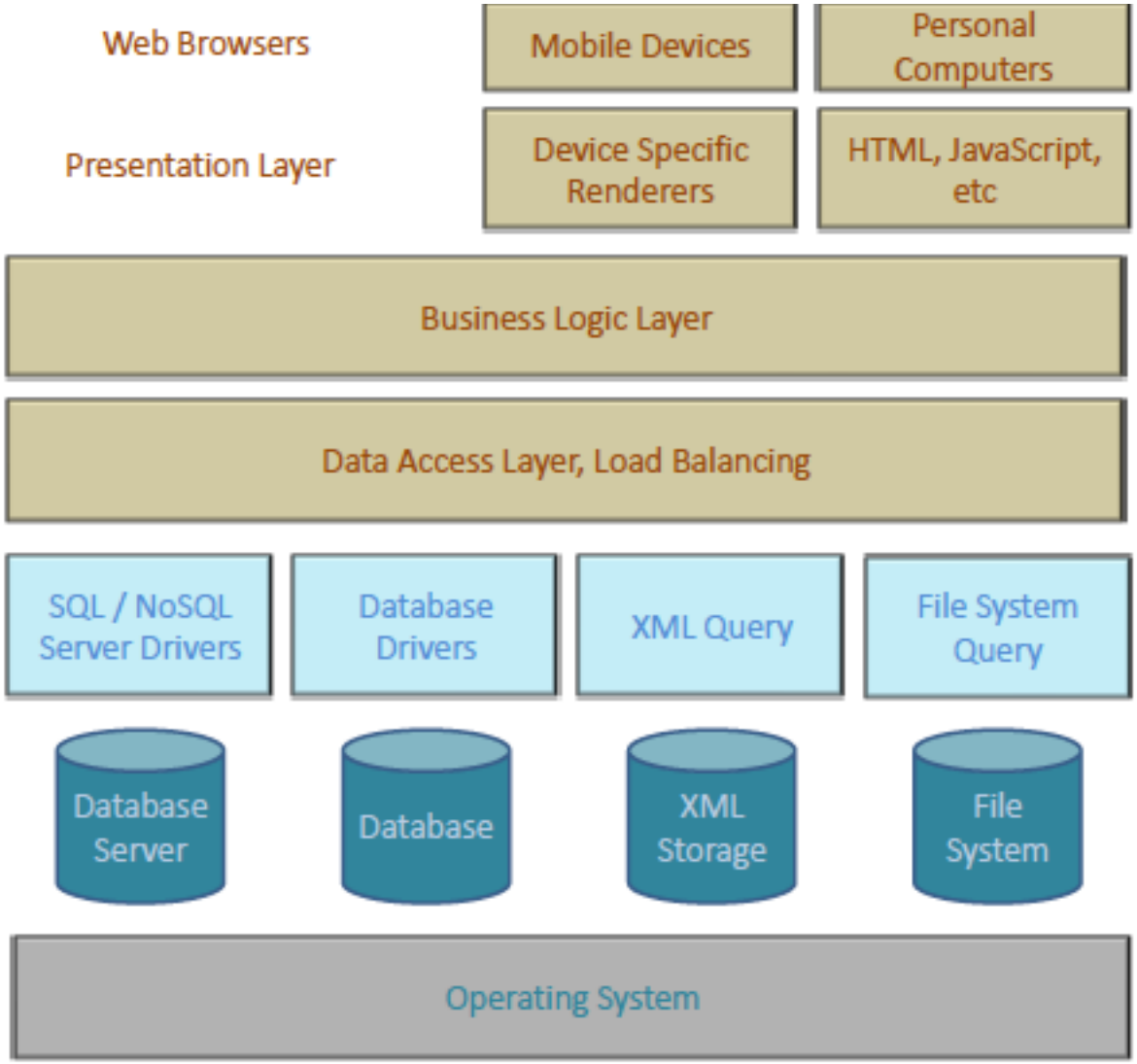


Relative Power Accuracy for CPU2006





Performance Proxies for Big Data Workloads

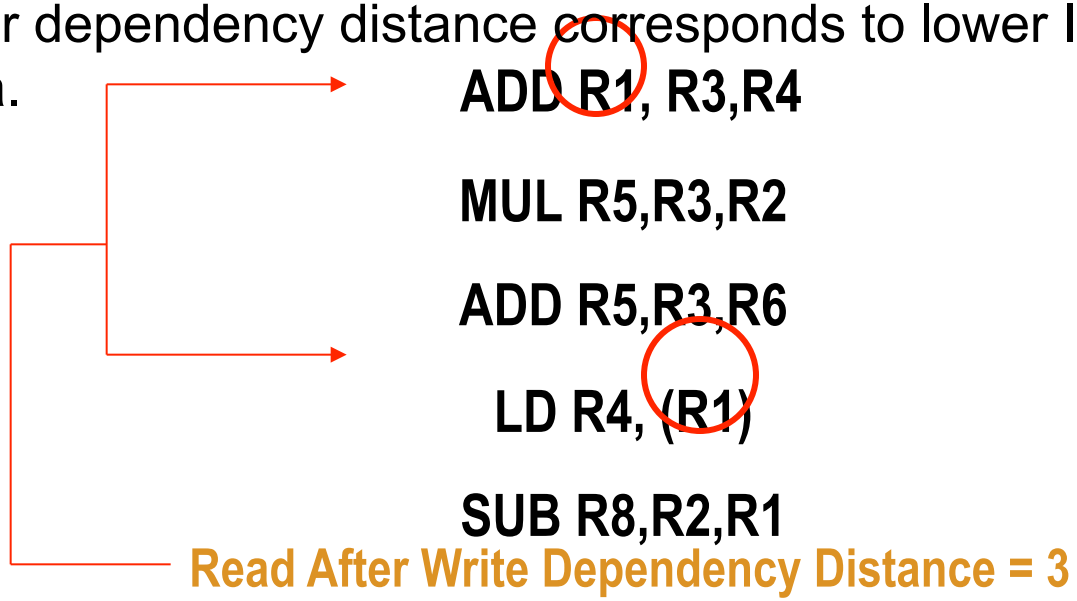


Instruction-level Profiling Impractical; Estimate METRICS from Perf Counters

- **Instruction-mix**
  - Measured using hardware performance counters
  - Fraction of integer (INT) ALU, INT MUL, INT DIV, floating-point (FP) ADD, FP DIV, FP MUL, loads, stores, control-flow instructions.
- **Instruction footprint**
  - Derived based on the instruction cache miss rate of the original application on a default instruction cache size (64KB, 64B line-size, 2-way set-associative).
- **Average basic-block size**
  - Estimated based on the instruction count and control instruction mix metrics.

Instruction-level METRIC Estimation (Contd)

- **Instruction level parallelism:**
  - Modeled by controlling the dependency distance (DD) between instructions.
  - Lower dependency distance corresponds to lower ILP and vice versa.



  - Avg DD estimated based on the fraction of dependency-related stall events (measured using hardware performance counters) of an application.

Memory-access characteristics

- Original approach – PIN-based memory locality characterization
  - Local Stride - Per ld/st stride distribution
  - Global Stride – Stride distribution of the global memory access streams
- Perf Counter Approach: Strides estimated based on the data cache miss rates of the original application.
  - E.g., 50% hit-rates can be modeled using a stride of 8 (assuming 64B cache-line size)

CONTROL-FLOW BEHAVIOR

- Branch predictability (BP) of the original application is measured using the branch prediction rate.
- A branch instruction's predictability can be controlled using its transition frequency.

$$\text{Branch Transition Rate} = \frac{\#\ \text{of Taken—Not Taken transitions}}{\#\ \text{of times executed}}$$

- PerfProx estimates the fraction of control instructions in the proxy benchmark that will have a particular predictability behavior.

Find the set of f_i such that

$$BP = f_1 * 1 + f_2 * 0.5 + f_3 * 0.33 + \ldots$$

- Assuming a 2-bit saturating counter based branch predictor, f_1, f_2, f_3 … are the fraction of branches with 100% (TTTTTTTT or NNNNNNN), 50% (TNTNTNTN), 33% (TTNTTNTTN) predictability respectively.

System CALL Activity

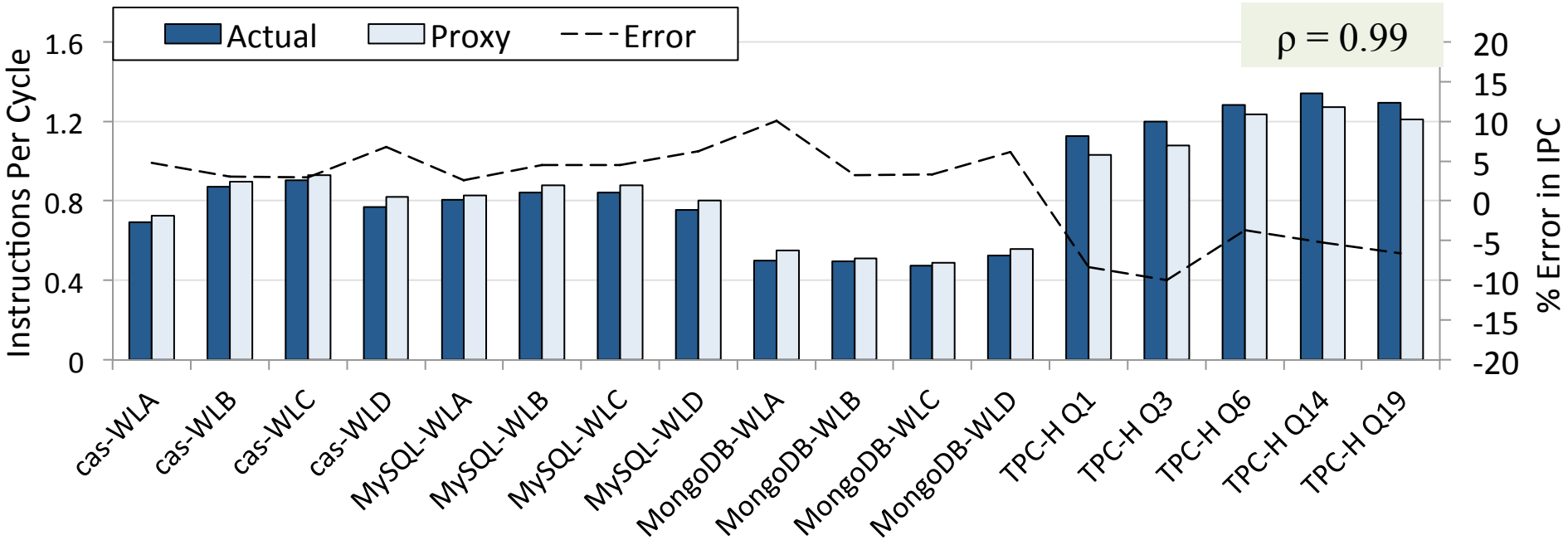
- Many emerging big-data applications spend a significant fraction of their execution time (upto 30%) executing operating system (OS) code.
- System activity monitored using *STRACE* tool and the fraction of user-mode and kernel-mode instructions using hardware performance counters.
- Modeled by inserting target fraction of system calls into the basic blocks in the proxy benchmark.

DATABASE PROXY EVALUATION

- Databases - Cassandra, MongoDB and MySQL
- Benchmarks
  - Data analytics (TPC-H Benchmarks) - 10GB, Full runs of Q1, Q3, Q6, Q14 and Q19.
  - Data serving (Yahoo! Cloud Serving Benchmark, YCSB) – 12GB, 100 Million operations.
- Performance monitoring performed on real systems using Linux Perf tool.
- Average instruction-count of the generated proxy benchmarks is ~2 billion (~520 times smaller than original database applications).

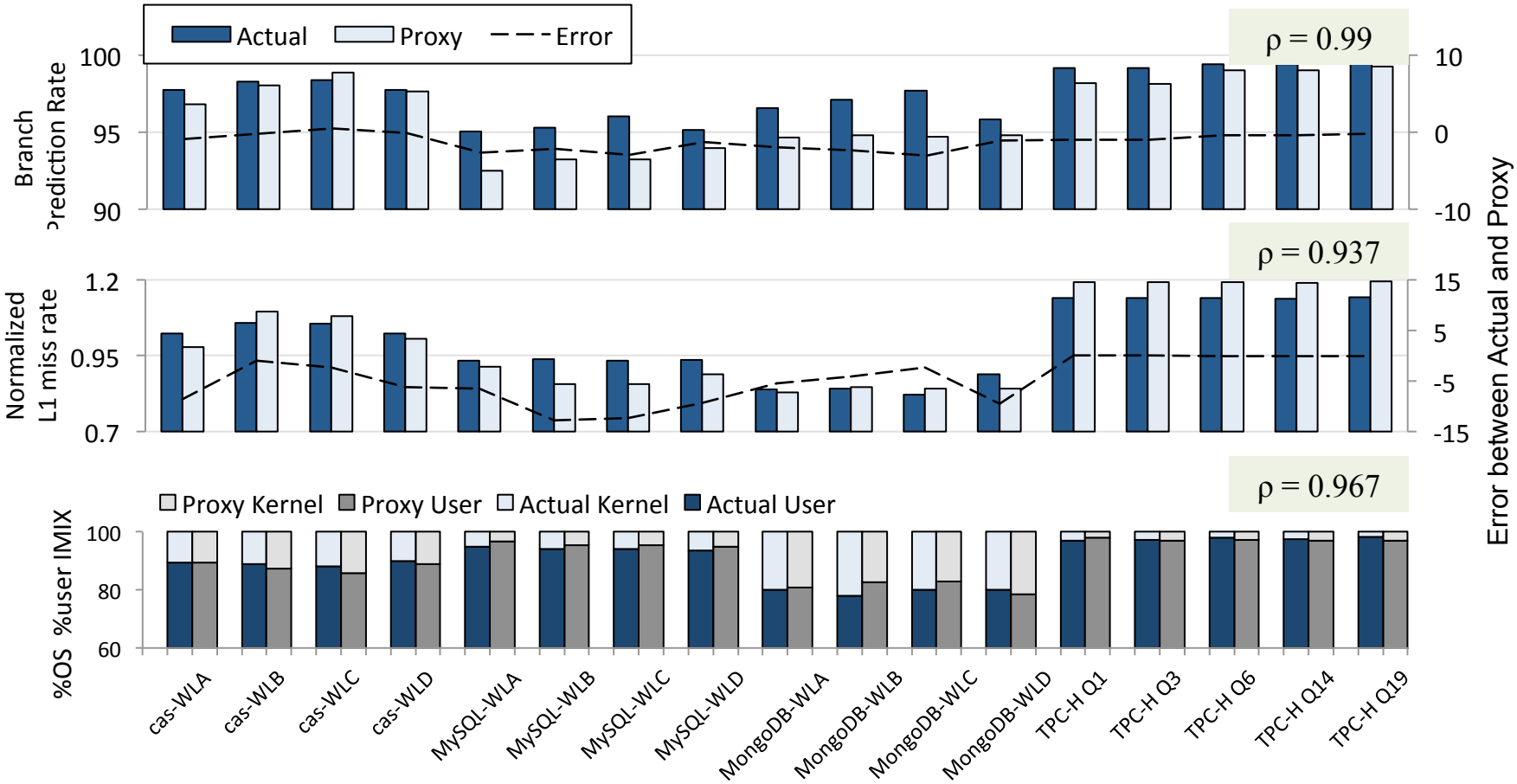
Configuration	System-A	System-B
Core Architecture	64-bit processor, Core micro-architecture	64-bit processor, Ivy-bridge micro-architecture
Core Frequency	2 GHz	2.50 GHz
Cache Configuration	Private L1 caches (64 KB I and D caches), 12 MB L2 cache	Three levels of caches, 1.5MB L2, 15MB L3 cache
Memory	16 GB DRAM	64 GB DRAM

Performance VALIDATION ON system-A (IPC)

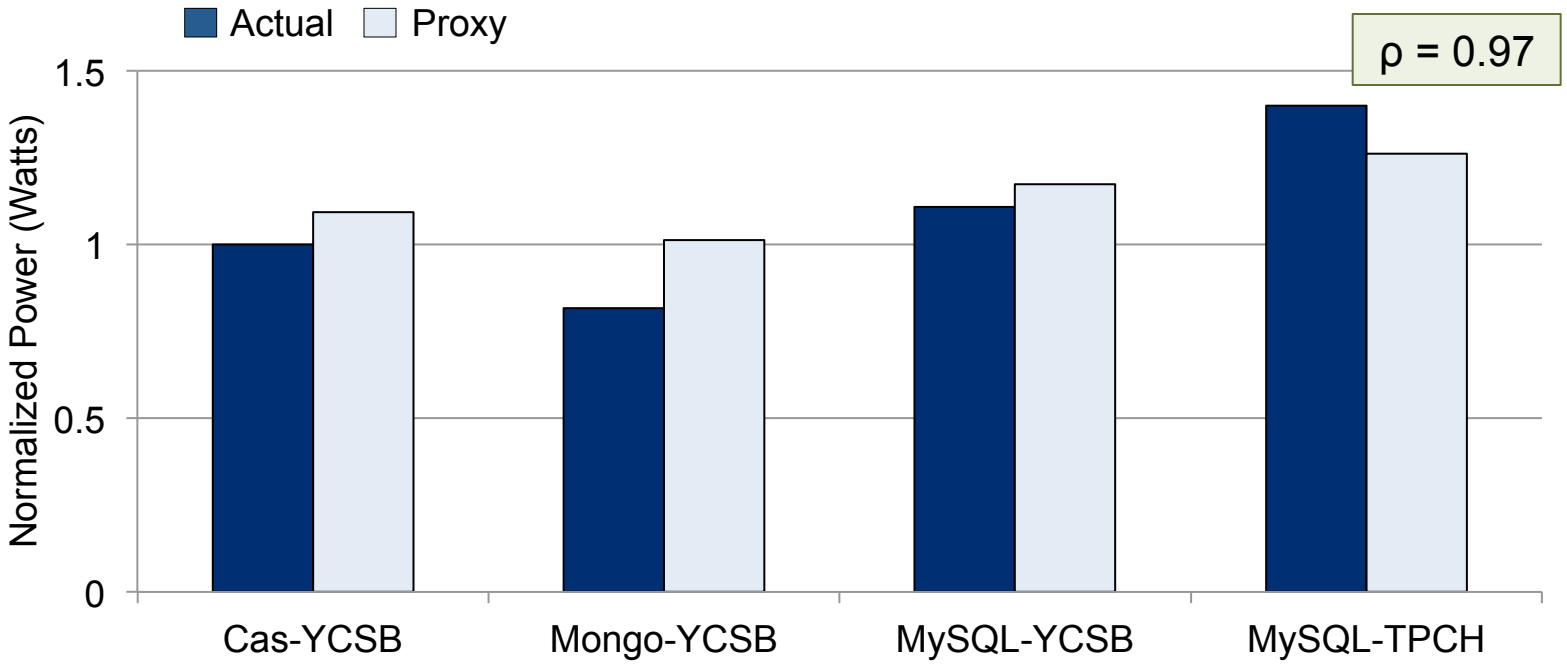


The proxy benchmarks mimic the IPC of the original applications with 94.9% (avg) accuracy for data-serving applications and 93.5% (avg) accuracy for data-analytics applications.

Performance VALIDATION ON system-A

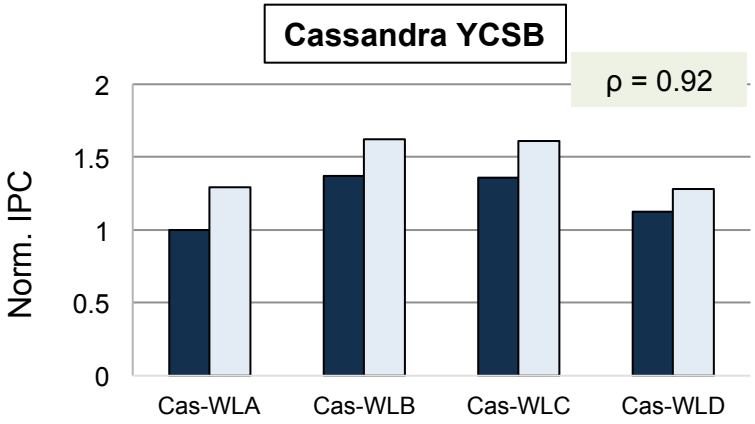


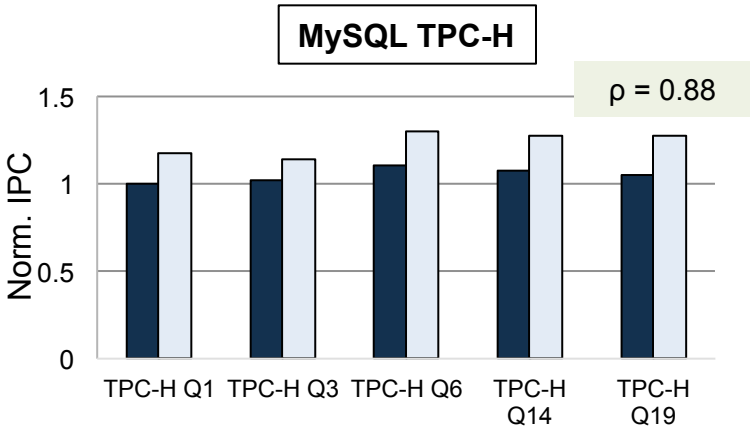
POWER VALIDATION ON system-A

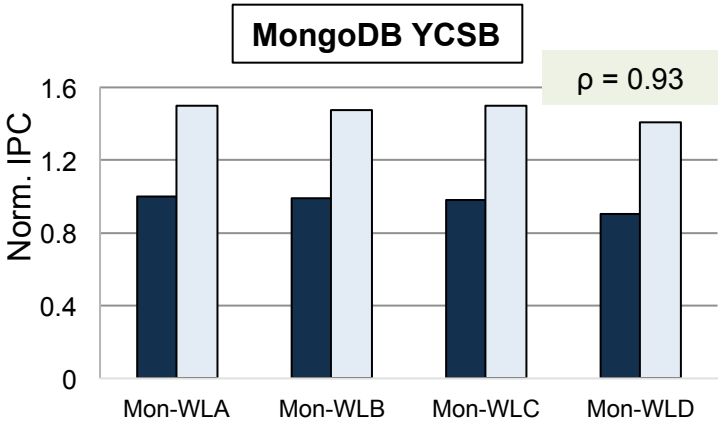


Correlation between the average power consumption of the proxy and actual applications is high (0.97).

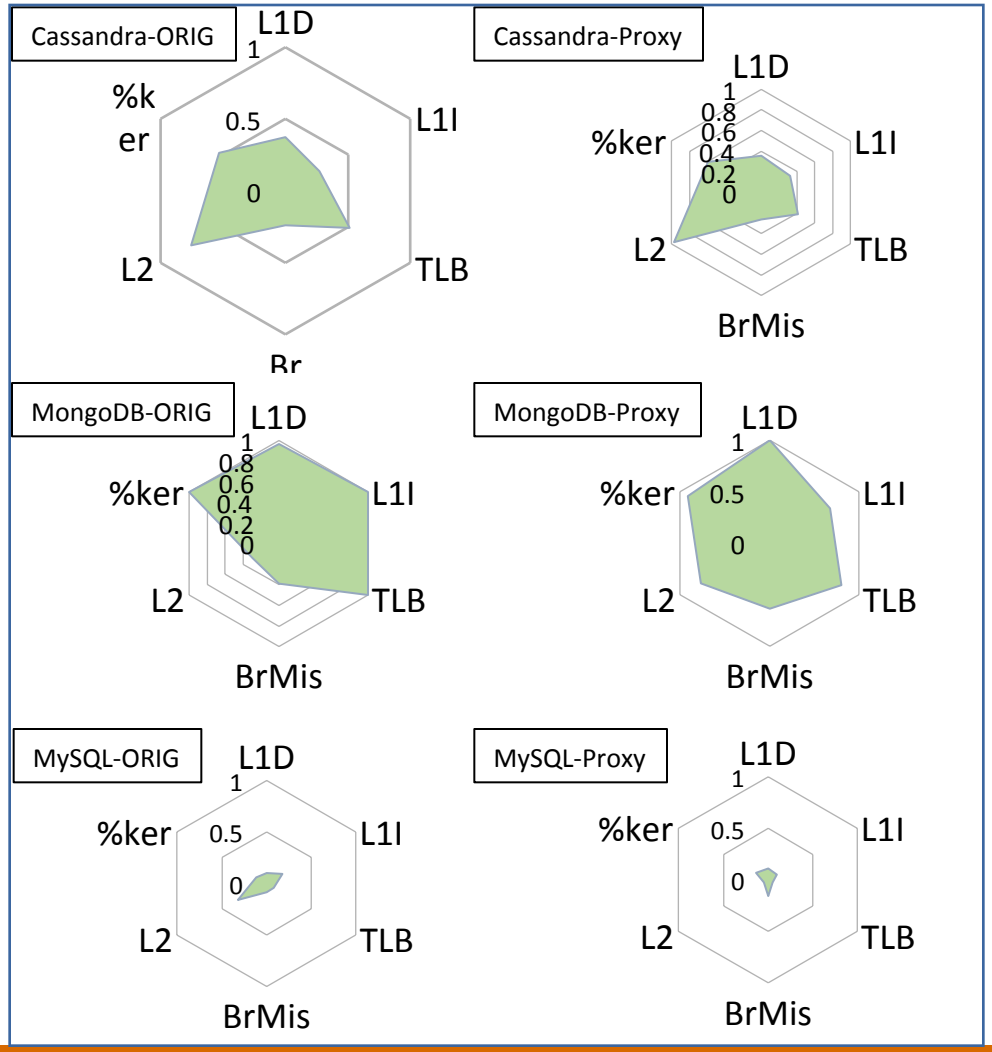
PERFORMANCE CROSS-VALIDATION ON system-B





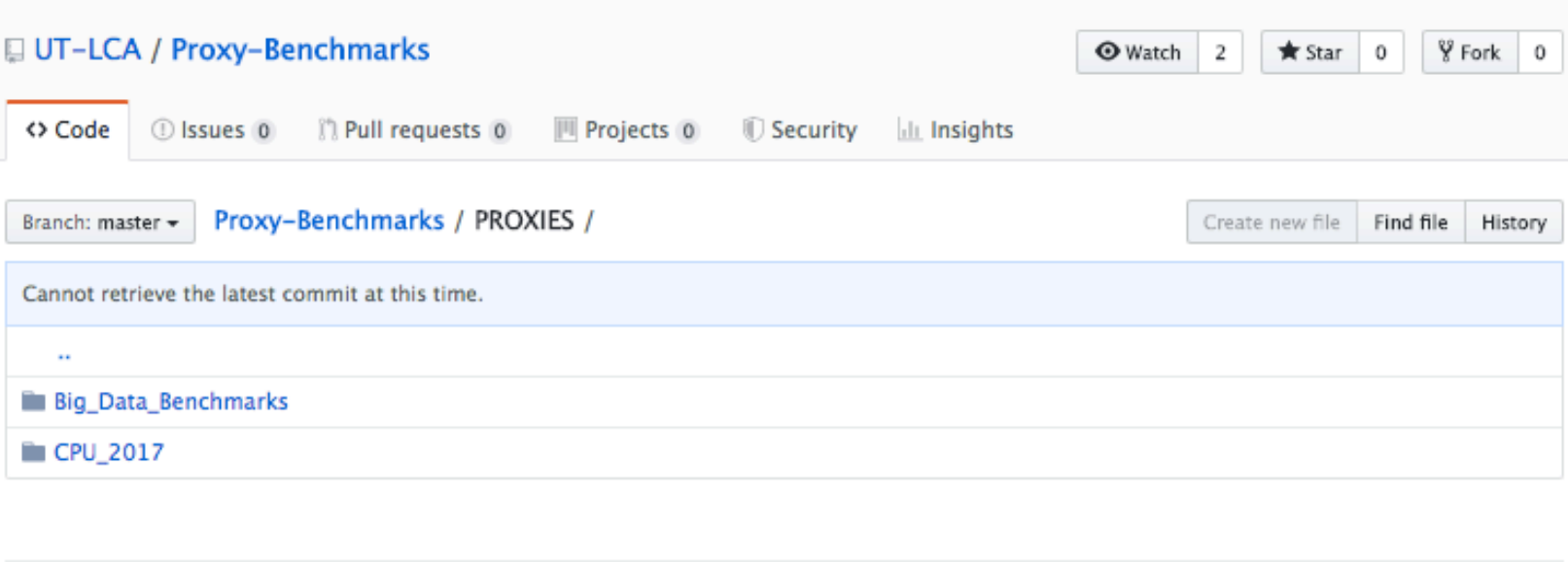


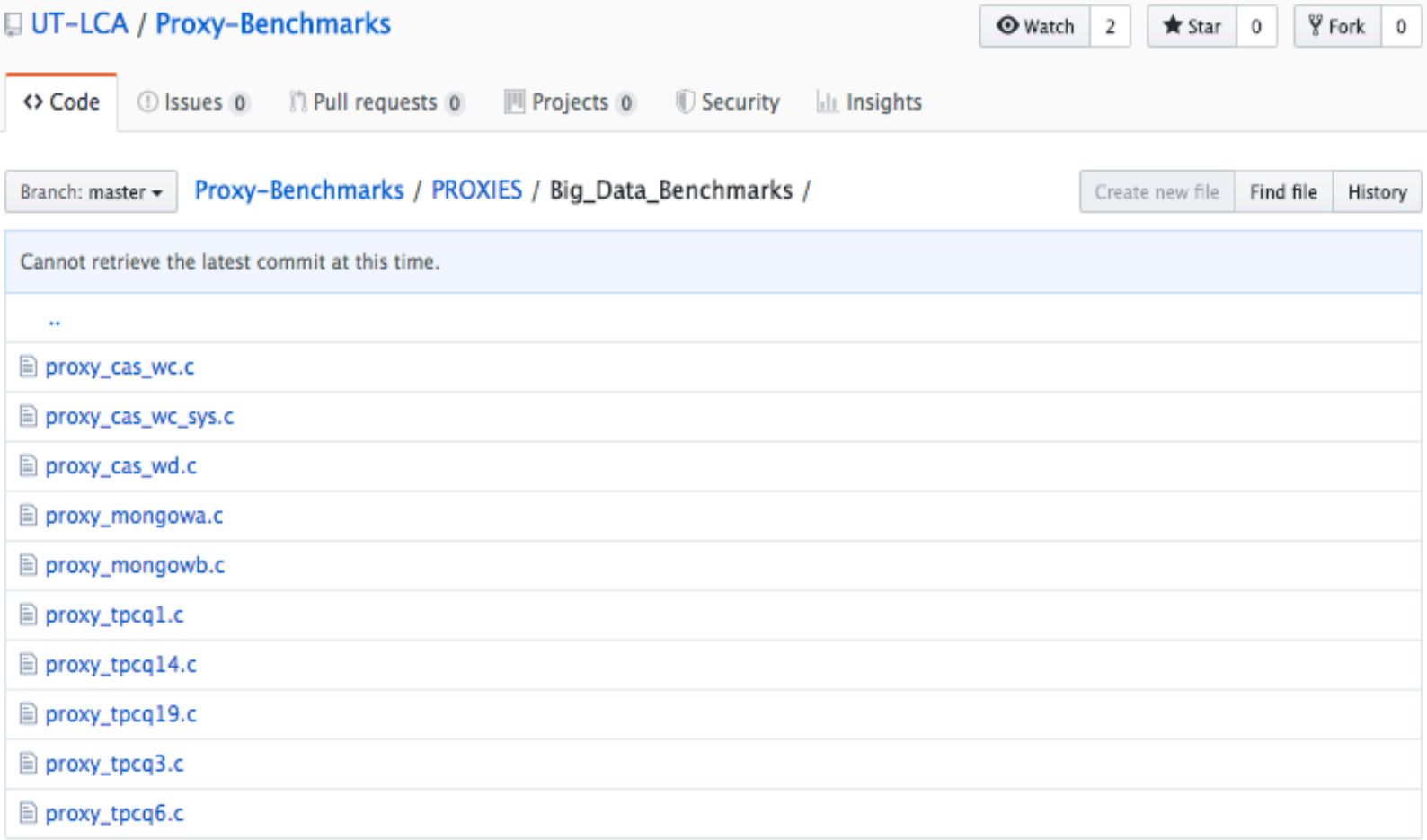
SUMMARY OF ORIGINAL TO PROXY COMPARISON



Where to find proxies

- **Github repository with most up to date proxies: https://github.com/UT-LCA/Proxy-Benchmarks**
- **Proxies are all c programs that can be compiled with GCC**
  - Example: gcc Deepsjeng.c –o Deepsjeng.o
- **Performance counters can be gathered with perf or other tools**
  - Example: perf Deepsjeng.o


UT-LCA / Proxy-Benchmarks
Watch 2
Star 0
Fork 0
Code
Issues 0
Pull requests 0
Projects 0
Security
Insights
Branch: master
Proxy-Benchmarks / PROXIES /
Create new file
Find file
History
Cannot retrieve the latest commit at this time.
..
Big_Data_Benchmarks
CPU_2017


UT-LCA / Proxy-Benchmarks
Watch 2
Star 0
Fork 0
Code
Issues 0
Pull requests 0
Projects 0
Security
Insights
Branch: master
Proxy-Benchmarks / PROXIES / Big_Data_Benchmarks /
Create new file
Find file
History
Cannot retrieve the latest commit at this time.
..
proxy_cas_wc.c
proxy_cas_wc_sys.c
proxy_cas_wd.c
proxy_mongowa.c
proxy_mongowb.c
proxy_tpcq1.c
proxy_tpcq14.c
proxy_tpcq19.c
proxy_tpcq3.c
proxy_tpcq6.c

UT-LCA / Proxy-Benchmarks

Watch 2 | Star 0 | Fork 0

Code | Issues 0 | Pull requests 0 | Projects 0 | Security | Insights

Branch: master ▾ Proxy-Benchmarks / PROXIES / CPU_2017 /

Create new file | Find file | History

Cannot retrieve the latest commit at this time.

..

- deepsjeng.c
- exchange2.c
- gcc.c
- leela.c
- mcf.c
- omnetpp.c
- perl.c
- x264.c
- xalancbmk.c
- xz.c

SPEC'17 BENCHMARK OVERVIEW

- CPU2017 Benchmark Overview
  - 43 Benchmarks: Speed & Rate versions of INT and FP programs
- CPU2017 vs CPU2006 Runtime Comparison

Runtime Chars.	SPEC CPU2017	SPEC CPU2006
Average Inst. Count	7,876,399,266,409	1,896,452,771,834
Total run time (all benchmarks)	72, 299 seconds	9860 seconds

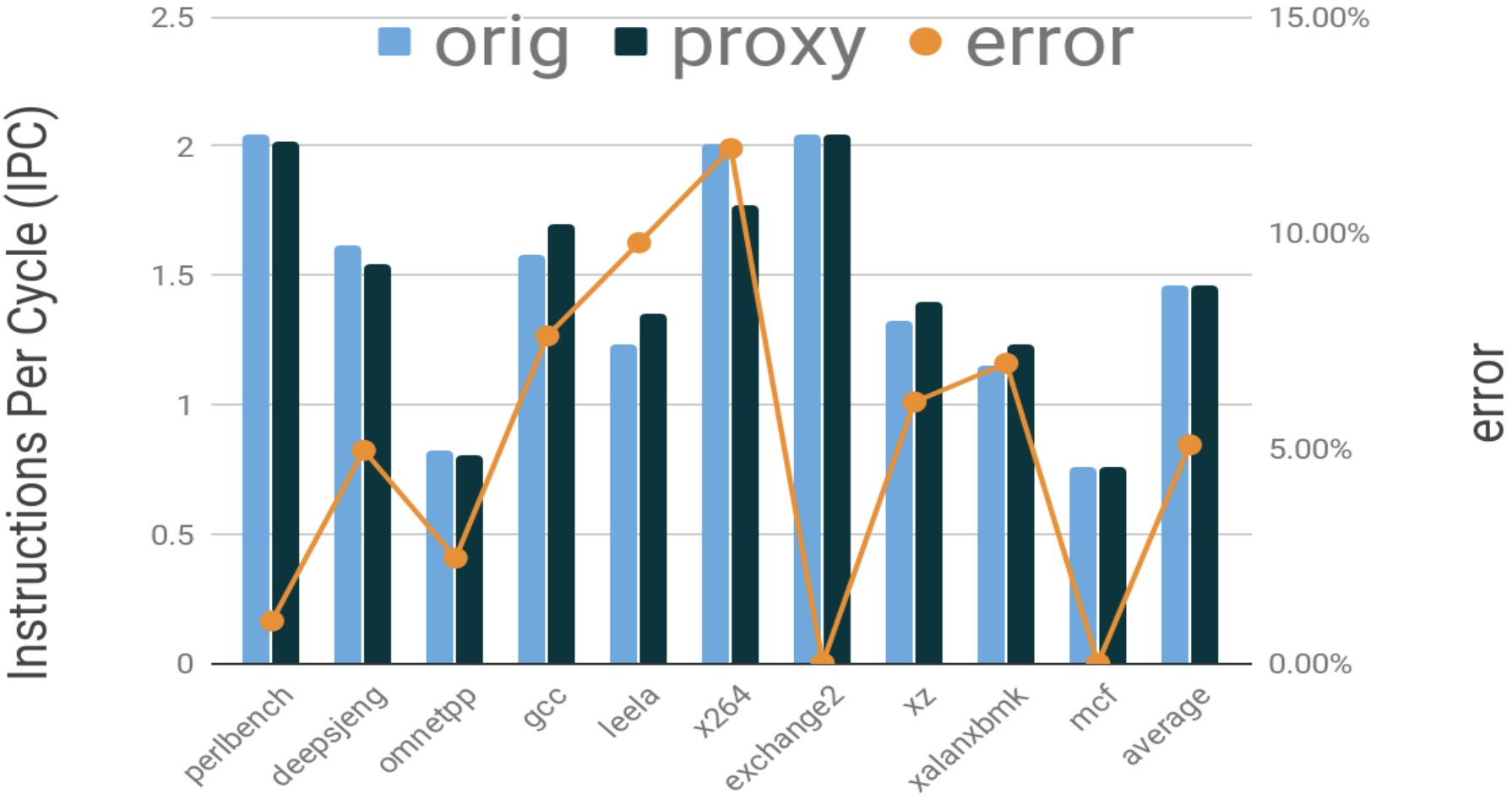
Benchmark Characteristics

Benchmark name	Language	KLoC	Application area
perlbench_s	C	362	Perl interpreter
Deepsjeng_s	C++	10	Artificial intelligence: alpha beta tree search (chess)
Omnetpp_s	C++	134	Discrete event simulation – computer network
Gcc_s	C	1304	GNU C compiler
Leela_s	C++	21	Artificial intelligence: Monte carlo tree search (Go)

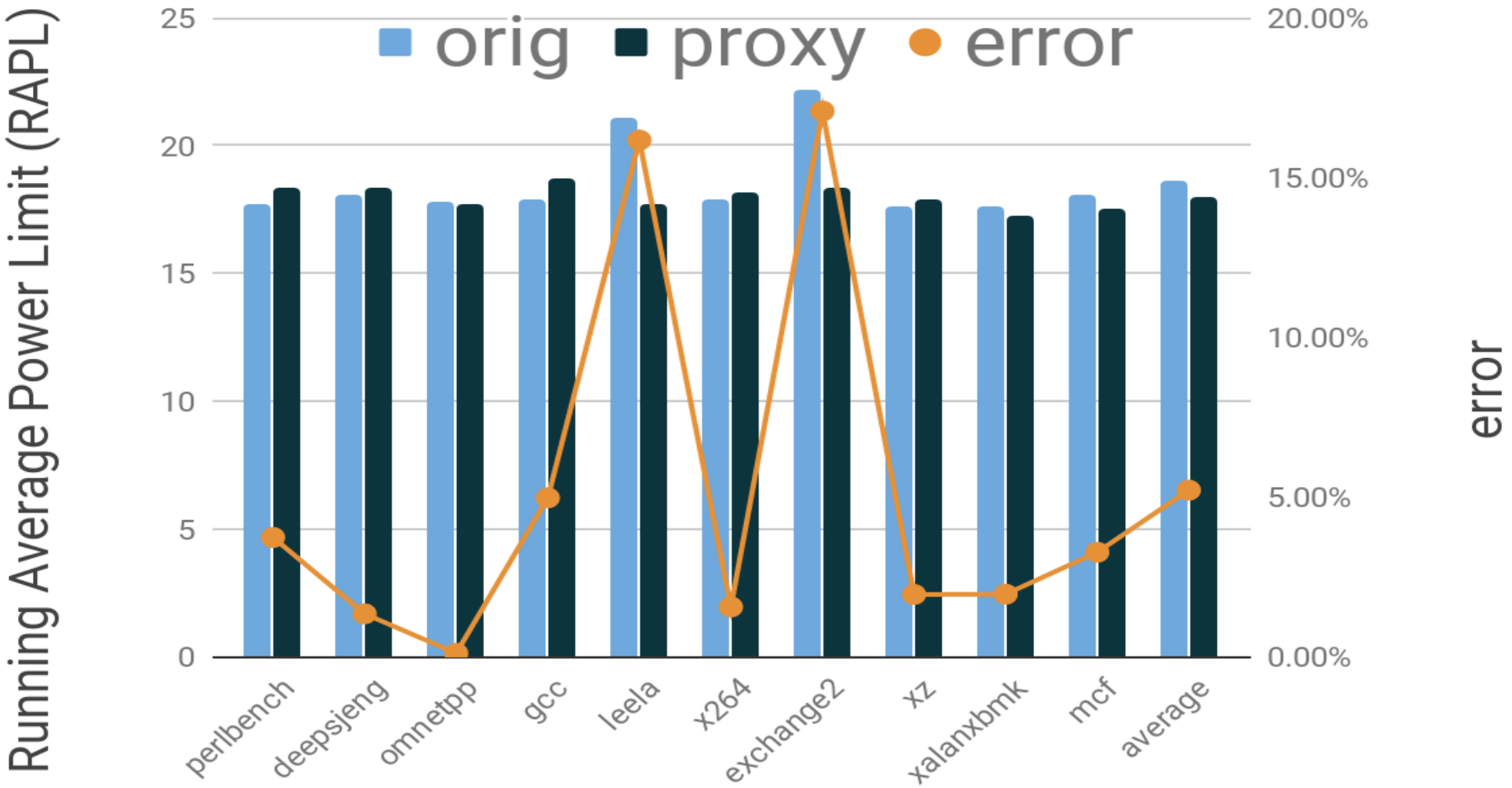
Benchmark Characteristics

Benchmark name	Language	KLoC	Application area
xalanxbmk_s	C++	520	XML to HTML conversion via XLST
mcf_s	C	3	Route Planning
x264_s	C	96	Video compression
exchange2_s	Fortran	1	Artificial Intelligence: recursive solution generator (Sudoku)
xz_s	C	33	General Data Compression

Proxy Performance (IPC)



Proxy Performance (Power)



Degree of Miniaturization

	perlbench		deepsjeng		omnetpp	
	Orig	Proxy	Orig	Proxy	Orig	Proxy
#Dyn Instructions	1,283,859,685,081	630,103,279	2,275,616,966,086	671,753,323	1,102,787,632,966	608,628,244
Orig/Proxy # Instructions	**2038**		**3388**		**1812**	

	gcc		leela			
	Orig	Proxy	Orig	Proxy		
#Dyn Instructions	4,577,230,000,000	982,597,009	2,245,850,995,706	423,060,220		
Orig/Proxy # Instructions	**4658**		**5309**			

- **all proxies have less than 1 billion dynamic instruction count**

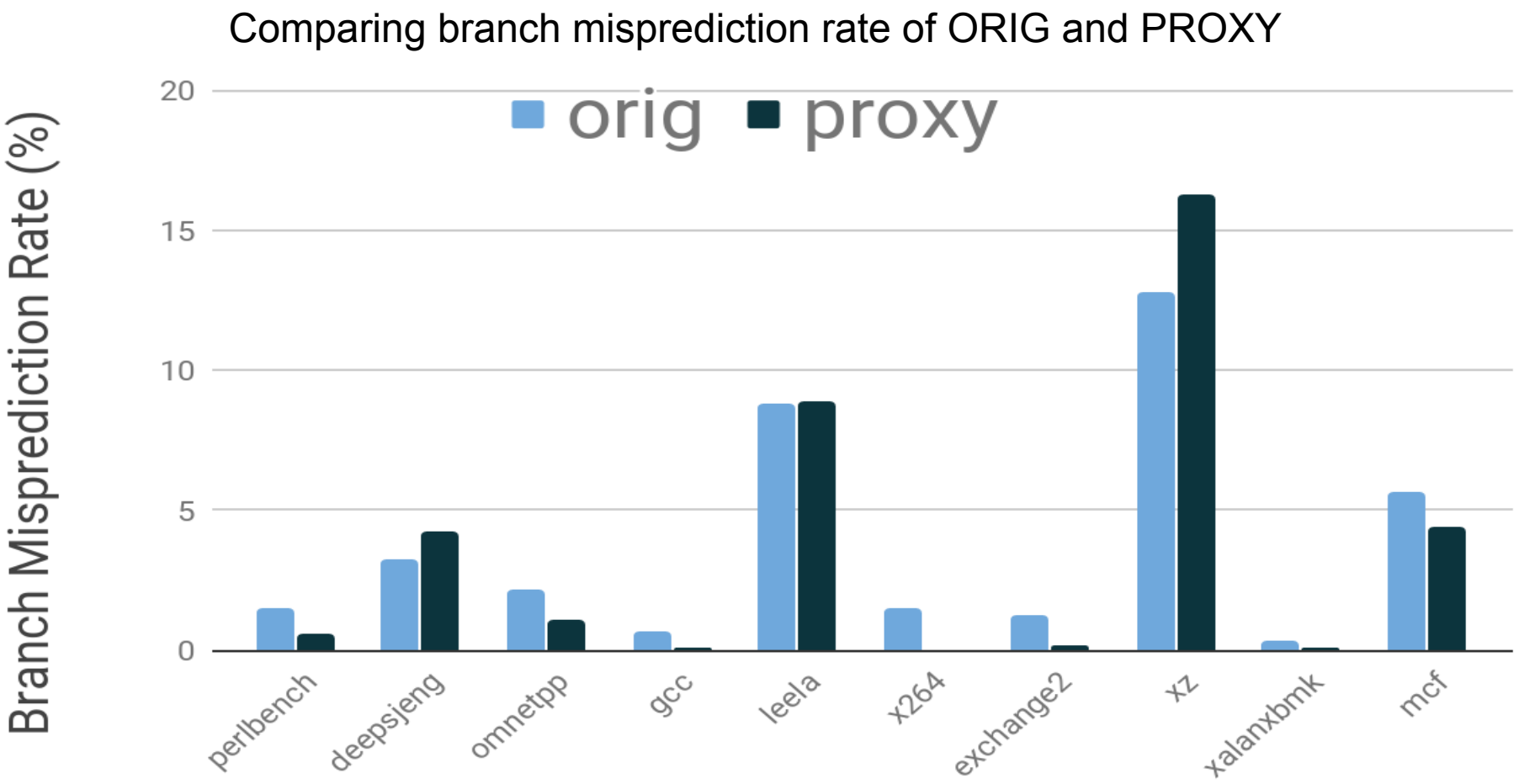
Degree of Miniaturization

	xalanxbmk		mcf		x264	
	Orig	Proxy	Orig	Proxy	Orig	Proxy
#Dyn Instructions	1,315,970,803,200	510,531,529	1,782,190,000,000	789,554,535	5030,000,000,000	809,326,203
Orig/Proxy # Instructions	**2578**		**2257**		**6219**	

	exchange2		xz			
	Orig	Proxy	Orig	Proxy		
#Dyn Instructions	6,643,720,000,000	895,456,832	5,030,530,000,000	818,316,812		
Orig/Proxy # Instructions	**7419**		**6147**			

- **all proxies have less than 1 billion dynamic instruction count**

Control-flow Behavior



Cache & TLB Performance (1/2)

	perlbench		deepsjeng		omnetpp		gcc		leela	
	ORIG	PROXY	ORIG	PROXY	ORIG	PROXY	ORIG	PROXY	ORIG	PROXY
DTLB MPKI	0.10	0.01	0.22	0.01	5.00	0.09	0.34	0.02	0.04	0.02
Dcache MPKI	3.51	4.77	4.55	7.30	36.64	32.88	12.66	10.05	4.98	4.25
L2 MPKI	1.64	0.00	1.41	0.02	17.79	13.74	1.38	4.99	0.58	0.01
L3 MPKI	0.01	0.00	1.10	0.00	6.31	9.70	0.62	4.61	0.00	0.00

Cache & TLB Performance (2/2)

	X264		Exchange2		xz		xalanxbmk		mcf	
	ORIG	PROXY	ORIG	PROXY	ORIG	PROXY	ORIG	PROXY	ORIG	PROXY
DTLB MPKI	0.03	0.01	0.00	0.02	2.04	0.00	2.89	0.02	1.48	0.05
Dcache MPKI	1.44	0.58	0.02	1.38	16.06	13.83	45.39	21.61	84.89	49.96
L2 MPKI	0.34	0.33	0.00	0.74	5.00	7.41	19.08	10.67	53.74	31.57
L3 MPKI	0.10	0.15	0.00	0.00	0.87	0.00	0.39	5.73	14.71	13.74

SimPoint as an Alternative – UT LCA SimPoints

- **SIMPOINTS are popularly used with simulators to reduce simulation time.**
- **UT Laboratory for Computer Architecture has generated SimPoints for SPEC CPU 2017**
- **SimPoints for SPEC CPU 2017 are available at https://github.com/UT-LCA/Scalability-Phase-Simpoint-of-SPEC-CPU2017/releases**

UT-LCA / Scalability-Phase-Simpoint-of-SPEC-CPU2017

Watch 4 | Star 3 | Fork 2

<> Code | Issues 1 | Pull requests 0 | Projects 0 | Security | Insights

Branch: master ▾ Scalability-Phase-Simpoint-of-SPEC-CPU2017 / SimPoints / Create new file | Find file | History

Cannot retrieve the latest commit at this time.

..

README.md

README.md

SimPoints

This folder holds the results of SimPoint study. Pinballs for SimPoints are available for download here.

Usage

Pinballs can be run natively using the Software Develop Emulator from Intel (https://software.intel.com/en-us/articles/intel-software-development-emulator). Gem5 (http://gem5.org/Simpoints) and Sniper (http://snipersim.org/w/Pinballs) also support the replaying of pinballs.

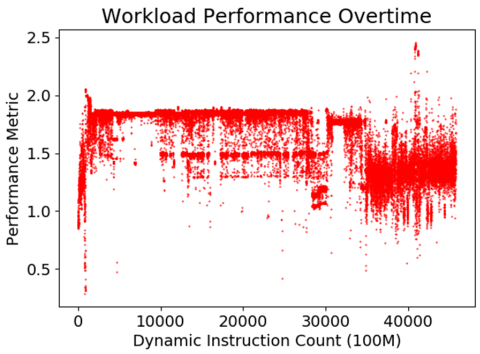
Interpreting the meaning of the 10 to 30 Pinballs for each workload in SPEC CPU 2017 is fairly straightforward. Each pinball base name specifies the key details for the pinball. An example is included below: deepsjeng.test_38307_t0r1_warmup1501_prolog0_region100000038_epilog0_001_0-05045.0.address

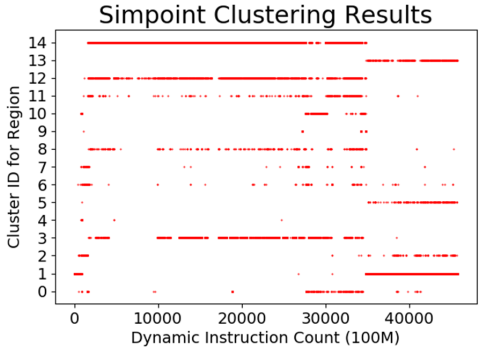
The file has many parts to its name, each with a specific meaning.

"program name"."run name"_"run number"_t"thread number"r"region number"_warmup"num warm up instructions"_prolog"num prolog instructions"*region"num instructions in region of interest"epilog"num epilog instructions""region number again"*"thread number again"-"weight of the region".0."file type"

SimPoint as an Alternative

- **Simpoint [1] breaks a workload into equal sized regions (100 Million Instructions)**
- **Regions are profiled based on micro-architecture independent Basic Block Vectors (BBV)**
- **Similar regions are clustered together based on the BBV using K-means**
- **A single region is simulated to represent each cluster**





Cluster ID	Cluster Weight	Representative Perf. Metric
0	0.87%	1.60
1	23.62%	1.21
2	1.36%	1.91
3	12.00%	1.47
4	0.09%	1.04
5	0.44%	1.47
6	0.69%	1.61
7	0.36%	1.46
8	0.84%	1.54
9	0.17%	1.33
10	3.60%	1.16
11	0.82%	1.74
12	5.28%	1.77
13	1.21%	1.41
14	48.65%	1.85

Weighted Sum

Simpoint Estimate
1.61

[1] Automatically Characterizing Large Scale Program Behavior: https://dl.acm.org/citation.cfm?id=605403

SimPoint Limitations

- **Dominant region can vary significantly from average behavior**

Cluster ID	Cluster Weight	Representative Perf. Metric
0	0.87%	1.60
1	23.62%	1.21
2	1.36%	1.91
3	12.00%	1.47
4	0.09%	1.04
5	0.44%	1.47
6	0.69%	1.61
7	0.36%	1.46
8	0.84%	1.54
9	0.17%	1.33
10	3.60%	1.16
11	0.82%	1.74
12	5.28%	1.77
13	1.21%	1.41
14	48.65%	1.85

SimPoints for CPU 2017

Benchmarks	Simulation Points	90 percentile Points	Instructions (billions)
SPECspeed Integer			
600.perlbench_s	25	13	2696
602.gcc_s	15	5	7226
605.mcf_s	21	11	1775
620.omnetpp_s	8	5	1102
625.x264_s	14	9	12546
631.deepsjeng_s	20	16	2250
641.leela_s	29	21	2245
648.exchange2_s	19	15	6643
657.xz_s	21	15	8264
Int Average	19.1	12.22	4607
SPECspeed Floating-point			
603.bwaves_s	9	3	66395
619.lbm_s	8	4	4416
638.imagick_s	15	6	66788
644.nab_s	12	5	13489
649.fotonik3d_s	17	7	4280
FP Average	12.2	5	24204

Multiple SimPoints

Benchmarks	Start Instructions(100 million)/Weight(%)
600.perlbench_s	(214681/28.85), (924424/12.60), (1277200/8.97), (1327590/7.96), (58779/7.77), (54633/5.43), (521547/5.39), (1233410/3.88), (752064/2.42), (701679/2.40), (789265/2.19), (1258340/2.06), (236975/1.78), (48882/1.71), (482810/1.44), (467463/0.77), (478321/0.68), (29698/0.68), (199580/0.61), (607466/0.55), (236842/0.54), (1339730/0.53), (1044350/0.30), (790578/0.30), (1141890/0.23)
602.gcc_s	(23077/48.65), (35817/23.62), (21069/12.00), (2812/5.28), (29282/3.60), (1353/1.36), (35911/1.21), (34118/0.87), (24369/0.84), (33794/0.82), (33733/0.69), (36933/0.44), (27933/0.36), (27185/0.17), (910/0.09)
605.mcf_s	(5943/21.24), (16201/12.54), (6315/10.82), (14440/10.75), (587/7.69), (6563/6.97), (13680/6.04), (7430/5.14), (14107/3.40), (12030/3.36), (12099/2.41), (9255/2.25), (7606/2.12), (14669/2.02), (7843/0.93), (15653/0.89), (12189/0.50), (17489/0.42), (9375/0.27), (15180/0.19), (1765/0.07)
620.omnetpp_s	(6053/32.56), (6737/30.32), (2966/15.25), (623/9.00), (1387/6.55), (9548/5.76), (10990/0.51), (10994/0.06)
625.x264_s	(49503/16.54), (22073/14.94), (3910/12.74), (42644/12.32), (10435/12.03), (46339/10.83), (26034/5.69), (41393/4.04), (11557/4.00), (24352/3.17), (4650/1.17), (11998/1.08), (53156/0.96), (31753/0.47)
631.deepsjeng_s	(17909/8.84), (21851/7.87), (84/7.02), (9700/7.00), (20151/6.98), (6204/6.37), (637/5.85), (7853/5.74), (20454/5.69), (15965/5.60), (14515/5.28), (9851/5.05), (3602/4.01), (12151/3.94), (5259/3.73), (11812/2.92), (11347/2.76), (3900/2.53), (15357/2.39), (0/0.46)
641.leela_s	(13198/8.63), (10150/7.15), (9272/6.85), (9870/6.75), (10635/6.57), (6871/5.64), (12252/5.49), (19208/4.29), (7341/4.17), (1830/3.90), (17685/3.67), (6004/3.59), (4934/3.45), (17328/3.37), (4952/3.10), (923/2.66), (21528/2.52), (1463/2.50), (22193/2.40), (1105/2.03), (820/1.91), (12296/1.84), (13929/1.60), (15764/1.47), (1193/1.36), (6041/1.15), (21296/0.98), (3514/0.96), (4/0.04)
648.exchange2_s	(41189/9.81), (46142/9.62), (28949/9.02), (32593/7.77), (46603/7.71), (11758/7.30), (50970/7.10), (27060/6.76), (3146/6.75), (66198/4.31), (38948/3.81), (17591/3.70), (31131/3.13), (20553/3.04), (9923/2.98), (56692/2.95), (424/1.68), (5824/1.63), (10190/0.95)
657.xz_s	(36567/13.00), (16895/9.39), (1400/8.92), (39442/8.76), (38731/6.97), (31742/6.59), (35941/6.33), (6913/5.80), (25357/5.25), (28962/4.09), (35764/3.86), (22587/3.21), (10870/3.02), (45973/2.94), (29995/2.88), (39598/2.72), (45675/2.26), (40484/2.04), (38264/1.88), (18/0.07), (44672/0.02)
603.bwaves_s	(59291/71.46), (248007/14.61), (287430/5.89), (313019/3.23), (315053/2.21), (95707/1.67), (318138/0.34), (263763/0.31), (303136/0.28)
619.lbm_s	(7134/69.69), (10742/14.92), (43825/4.43), (11740/4.36),(6240/3.65), (9400/2.38),(8491/0.56), (0/0.01)
638.imagick_s	(551856/43.83), (394592/27.24), (467091/8.12), (28460/4.97), (91452/4.24), (4209/2.76), (41027/2.50), (515828/2.45), (72623/1.27), (656007/1.21), (651005/0.78), (10504/0.54), (102890/0.04), (272722/0.04), (567401/0.04)
644.nab_s	(9237/34.24), (52046/22.75), (56855/17.91), (58776/9.76), (122599/7.45), (133757/6.91),(48950/0.26), (120710/0.25), (130263/0.16), (104620/0.14), (76009/0.09), (84205/0.08)
649.fotonik3d_s	(19088/43.91), (2468/29.02), (56395/6.90), (1608/4.72), (11626/2.65), (53620/2.21), (17985/2.12), (33097/1.95), (35644/1.20), (38568/0.91), (31846/0.83), (44869/0.83), (3563/0.80), (46262/0.64), (33943/0.49), (43842/0.41), (36949/0.41)

How to read the Table of Multiple SimPoints

600.perlbench_s	(214681/28.85), (924424/12.60), (789265/2.19), (1258340/2.06), (236842/0.54), (1339730/0.53),
602.gcc_s	(23077/48.65), (35817/23.62), ((36933/0.44), (27933/0.36), (27
605.mcf_s	(5943/21.24), (16201/12.54), (6 (9255/2.25), (7606/2.12), (1466

In many cases, the dominant SimPoint has less than 10% weight

631.deepsjeng_s	(17909/8.84), (21851/7.87), (84/7.02), (9700/7.00), (20151/6.98), (62 (3602/4.01), (12151/3.94), (5259/3.73), (11812/2.92), (11347/2.76), (
641.leela_s	(13198/8.63), (10150/7.15), (9272/6.85), (9870/6.75), (10635/6.57), ((4934/3.45), (17328/3.37), (4952/3.10), (923/2.66), (21528/2.52), (14 (1193/1.36), (6041/1.15), (21296/0.98), (3514/0.96), (4/0.04)
648.exchange2_s	(41189/9.81), (46142/9.62), (28949/9.02), (32593/7.77), (46603/7.71) (17591/3.70), (31131/3.13), (20553/3.04), (9923/2.98), (56692/2.95),

Proxies allow us to make a single proxy for the whole benchmark

Or

One proxy per SimPoint,

giving more accuracy than the dominant SimPoint

Industry Use

IBM – 2006

AMD – 2010

Intel 2017-2019

Miniature Benchmarks for RTL Model Validations

- **"Automatic Testcase Synthesis and Performance Model Validation for High Performance PowerPC Processors", Bell, Bhatia, John, Stuecheli, Griswell, Tu, Capps, Blanchard, and Thai, IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). March 2006.**

Presentation by EMILY SHRIVER, Intel

Improving Accuracy of Proxies

- **Current Challenges**
  - **Major Inaccuracies**
  - **Branch Behavior Capture**
  - **Memory Behavior Capture**

Improving Branch Behavior

- **Branch Profiling has become more precise in order to increase accuracy**
  - **Initially Global Transition Rate [1]**
  - **Added Global Taken Rate**
  - **Added Per Branch (local) Transition and Taken Rate**
  - **Added inter Branch Correlation [2]**
  - **Adding Variable inter Branch Correlation**

[1] M. Haungs, P. Sallee, et al "Branch transition rate: a new metric for improved branch classification analysis," [HPCA2000]

[2] Pan, Shien-Tai, Kimming So, et al "Improving the accuracy of dynamic branch prediction using branch correlation." [ACM1992]

Improving Branch Behavior

- **Branch Profiling has become more precise in order to increase accuracy**
  - **Initially Global Transition Rate [1]**
  - **Added Global Taken Rate**
  - **Added Per Branch (local) Transition and Taken Rate**
  - **Added Inter-Branch Correlation [2]**
  - **Added Variable Inter-Branch Correlation**

[1] M. Haungs, P. Sallee, et al "Branch transition rate: a new metric for improved branch classification analysis," [HPCA2000]

[2] Pan, Shien-Tai, Kimming So, et al "Improving the accuracy of dynamic branch prediction using branch correlation." [ACM1992]

Proxy Branch Generation – Overview

- **Branch mispredictions reduce overall IPC**
  - Speculative work must be discarded
- **Proxies should have branch predictor performance similar to original**
- **Metric of Interest: MPKI**
  - Misses per thousand branch instructions

Profiling Branch Behavior

- **Profile local behavior, i.e. per-static-branch**
- **Maintain a data structure for each static branch in the workload**
- **Every time a branch is encountered:**
  - Increment the "encountered" count for the branch
  - If taken: increment the "taken" count for the branch
  - Compare taken behavior with the last time it was encountered
    - If different: increment the "transition" count for the branch
- **Write raw results to file**
- **Discard data from extremely rare branches**
  - 10% of static branches can make up 99.99% of dynamic branches

Profiling Branch Correlations

- **Need to track a history of global branch behavior**
  - Sliding window for last 512 global conditional branches
- **Maintain a 512-entry table for each static branch**
- **Every time a branch *b* is encountered:**
  - Compare "is_taken[b]" to all branches in window:
    - If is_taken[b] $\oplus$ History[i] then Corr[b][i] -= 1
    - Else Corr[b][i] += 1
- **Normalize values to range (-1, 1)**
  - Corr[b][i] /= Encountered[b]
- **Write to file; discard rare branches and weak correlations**

Proxy Branch Generation

- **Use original static branches as "templates" for proxy branches**
  - Each proxy branch matches the taken and transition rates of its template
- **How many branches should use each template?**
  - Each proxy branch is dynamically executed the same number of times
  - Particularly common original branches should be used for multiple proxy branches
  - Scaling step automatically performed

Ordering of Proxy Branches

- **Originally, we ordered branches randomly**
  - This worked well in *some* cases, but clobbered correlations
- **So: Order branches to minimize correlation error**
  - Currently, we just look at the strongest correlation
  - We also only consider the immediate previous branch
  - Error = $\sum$ abs(max(Corr[b_n]) – $\rho(b_n, b_{n-1})$) for n = 2, …, N
  - Likely NP-hard; we don't yet have a great heuristic
    - Still gives pretty good results!

Improving Memory Access Patterns

- **The proxies in the repository use a stride based memory model that captures a per instruction memory pattern**
  - Example: a load instruction accesses x3000, then x3008, x3010, x3018, ….
- **Recent research (HALO) has found better accuracy through modeling[1]:**
  - Intra-region stride locality
  - Inter-region reuse locality
- **Developed proxies that implements these two techniques into proxies**

[1] Panda, Reena, and Lizy K. John. "HALO: A Hierarchical Memory Access Locality Modeling Technique For Memory System Explorations." In Proceedings of the 2018 International Conference on Supercomputing, pp. 118-128. ACM, 2018.

HALO Memory Behavior

- **For intra-region locality, a Markov chain is created for each region of memory to track patterns within**
- **The probability of switching between regions is calculated for the inter-region reuse locality**
- **HALO combines the intra-region locality and inter-region reuse distance to create similar memory traces**

HALO Memory Behavior

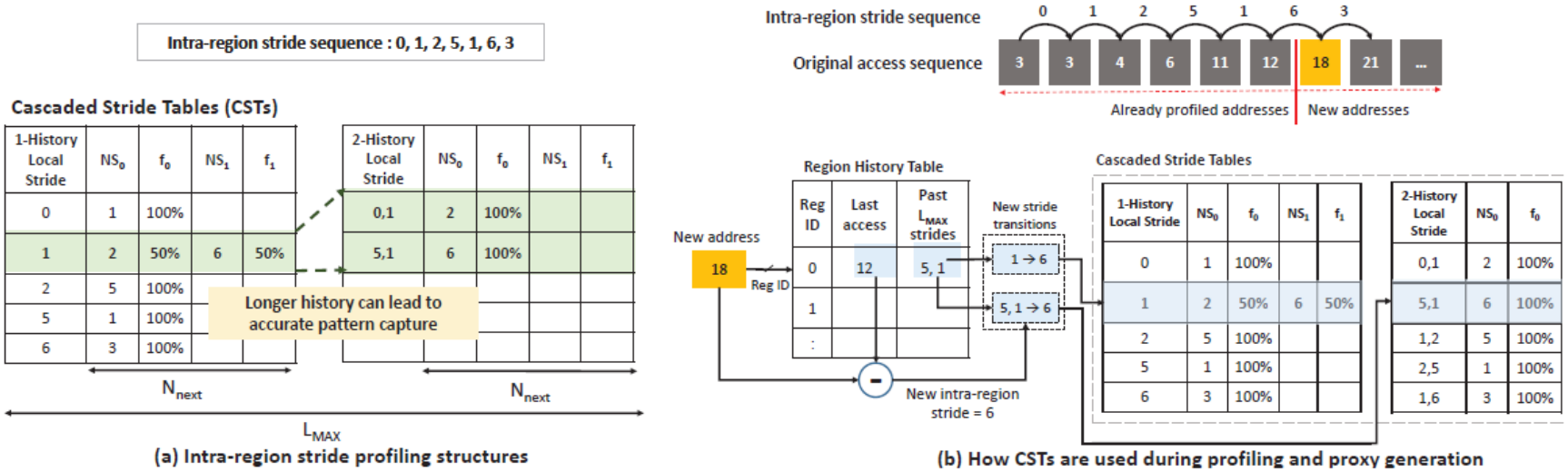


Figure 4: Intra-region locality profiling using cascaded stride tables (CSTs).

Leveraging Halo into Mind Prox

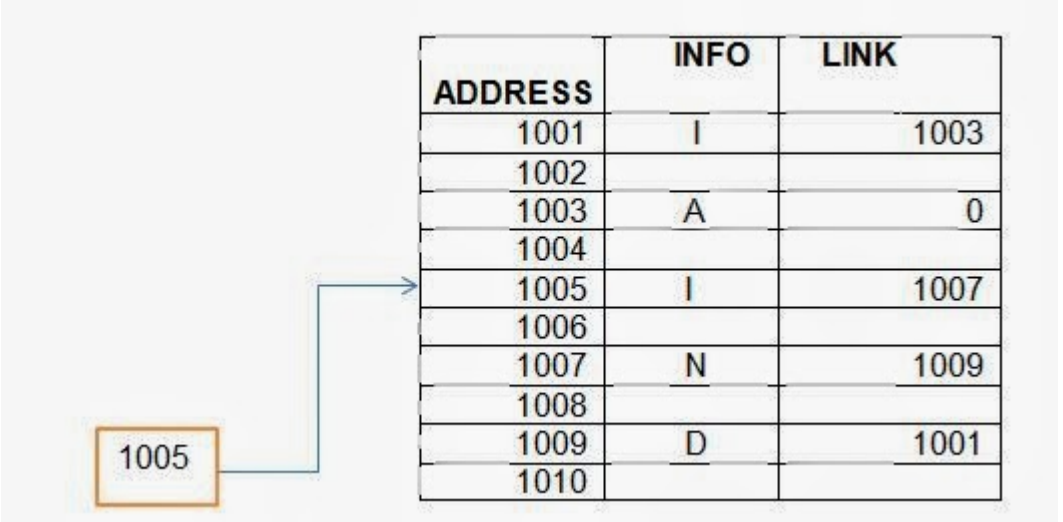
- **The Markov tables can be used to create a set of traces for each region**
  - This translates quite well to the large arrays already used in perfprox
- **The inter region reuse locality translates to which arrays to access in which order**
  - This can translate directly into accessing different arrays in an interleave that matches the Halo metrics
- **Embedding the trace into a proxy was the last challenge**

Challenge of Embedding Traces into Proxies

- **Straight forward solution is to have an array of memory locations and read from the array to get the next address**
  - This leads to an extra read to the array data structure
- **The main challenge is overhead, any proxy technique for memory cannot have extra reads to a data structure**
  - Why most proxies use stride-based techniques that slowly progress through memory

Linked List Solution

- **Linked lists allow opportunity for arbitrary memory behavior without overhead**
  - Walking a linked list updates the linked list pointer for every read
  - No useful information needs to be included in linked list, just the chain of pointers
  - Important step is converting a trace into a linked list format



ADDRESS	INFO	LINK
1001	I	1003
1002		
1003	A	0
1004		
1005	I	1007
1006		
1007	N	1009
1008		
1009	D	1001
1010		

Challenges with Linked Lists

- **Linked lists can only access an individual location once**
  - Many programs access locations repeatedly in a short period
- **Additionally, deciding how to deal with this issue leads to minimizing error between original trace and the linked list**
  - Accesses to same cache line can be seen as acceptable
- **Linked list can only handle reads, writes would clobber linked list**
  - Each node has a scratchpad for writing to instead of reading from
  - Thus writes are used to handle small offsets

Addressing the repeat access problem

- **Most accesses to the same location are successive**
  - From a stride perspective these would be approximately 0 offsets
- **Small offsets are removed from the linked list and treated as stride accesses in the proxy**
  - These would be memory accesses that do no update the linked list pointer
  - Currently all offsets less than 8 are removed
- **Issue is also addressed by shortening memory trace**

Reproducible Research

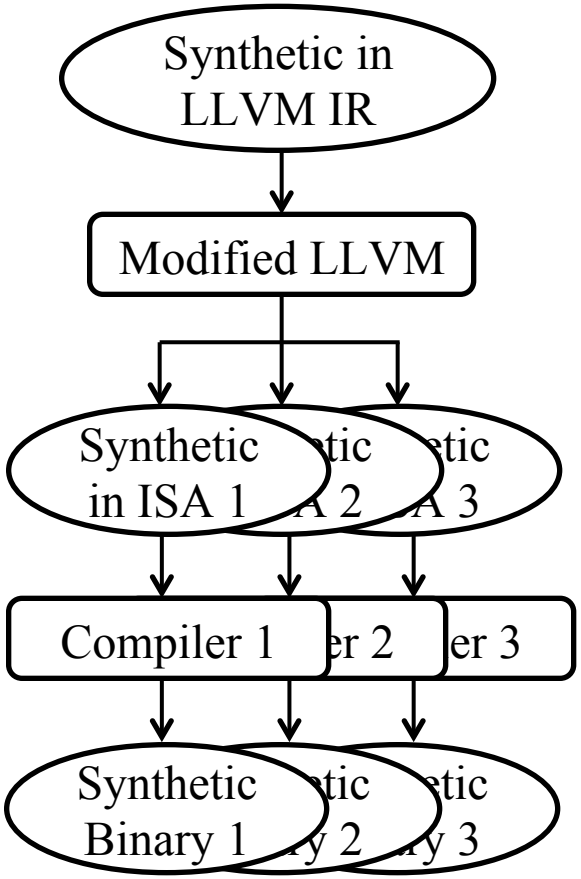
- We have developed a proxy generation methodology to generate miniature proxy benchmarks.
- Irrespective of the errors with their original counterparts, the proxies can be useful in reproducible research.
- Reproducibility is a problem in computer architecture research.
- Use these as benchmarks in architecture papers, and they can be used for validation when the next follow on work cannot fully simulate the earlier work completely due to simulator changes, runtime changes etc.

Reproducibility

- Reproducibility of Research using Proxies
  - Miniaturization Standpoint
  - Software Stack Abstraction Standpoint
  - Proprietary Codes Standpoint
  - Simulator Validation Standpoint
- Would like the research community to use the proxies along with other benchmarks
  - SPEC CPU 2017 Proxies
  - Data base (Cassandra, MongoDB, MySQL) Proxies
  - Github

ISA Independent Synthetic Benchmark Generation

- **Step 1: Generate synthetic code in LLVM IR**
  - **Use Static Single Assigned (SSA) form**
  - **Not bounded to specific ISA**
  - **Targeting total number of dynamic instructions to desired value**
- **Step 2: Create assembly for target ISAs**
  - **Use modified LLVM backend**
  - **Disable optimization paths to preserve characteristics**
- **Step 3: Compile binary for target ISAs**
  - **Use general compiler for each ISAs**
  - **Does not require specialized tools**



ISA Independent Synthetic Benchmarks

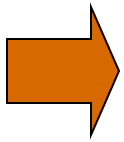
- **LLVM compiler infrastructure**
  - A virtual instruction set language
  - Various back-end support – x86, PowerPC, ARM, Alpha, SPARC, etc.
- **Use virtual ISA in SSA form**
  - Micro architecture independent
  - Avoid machine-specific optimizations in the front-end
  - Back-end takes care of machine-dependent optimizations

```c
#include <stdio.h>

int main()
{
    int i;
    int q,r,t, x[100000],  y[100010], z[100011];

    for (i=0;i<100000;i=i+1)
    {
        x[i]= q + y[i]*(r*z[i+10] + t *z[i+11]);
        printf("%d\n",x[i]);
    }

    return 0;
}
```



```
%tmp1 = load i32* %i, align 4
%tmp2 = load i32* %i, align 4
%tmp3 = getelementptr [100010 x i32]* %y, i32 0, i32 %tmp2
%tmp4 = load i32* %tmp3, align 4
%tmp5 = load i32* %i, align 4
%tmp6 = add i32 %tmp5, 10
%tmp7 = getelementptr [100011 x i32]* %z, i32 0, i32 %tmp6
%tmp8 = load i32* %tmp7, align 4
%tmp9 = load i32* %r, align 4
%tmp10 = mul i32 %tmp8, %tmp9
%tmp11 = load i32* %i, align 4
%tmp12 = add i32 %tmp11, 11
%tmp13 = getelementptr [100011 x i32]* %z, i32 0, i32 %tmp12
%tmp14 = load i32* %tmp13, align 4
%tmp15 = load i32* %t, align 4
%tmp16 = mul i32 %tmp14, %tmp15
%tmp17 = add i32 %tmp10, %tmp16
%tmp18 = mul i32 %tmp4, %tmp17
%tmp19 = load i32* %q, align 4
%tmp20 = add i32 %tmp18, %tmp19
```

ISA Independent Synthetic Benchmarks

LLVM ISA

```
%tmp1 = load i32* %i, align 4
%tmp2 = load i32* %i, align 4
%tmp3 = getelementptr [100010 x i32]* %y, i32 0, i32 %tmp2
%tmp4 = load i32* %tmp3, align 4
%tmp5 = load i32* %i, align 4
%tmp6 = add i32 %tmp5, 10
%tmp7 = getelementptr [100011 x i32]* %z, i32 0, i32 %tmp6
%tmp8 = load i32* %tmp7, align 4
%tmp9 = load i32* %r, align 4
%tmp10 = mul i32 %tmp8, %tmp9
%tmp11 = load i32* %i, align 4
%tmp12 = add i32 %tmp11, 11
%tmp13 = getelementptr [100011 x i32]* %z, i32 0, i32 %tmp12
%tmp14 = load i32* %tmp13, align 4
%tmp15 = load i32* %t, align 4
%tmp16 = mul i32 %tmp14, %tmp15
%tmp17 = add i32 %tmp10, %tmp16
%tmp18 = mul i32 %tmp4, %tmp17
%tmp19 = load i32* %q, align 4
%tmp20 = add i32 %tmp18, %tmp19
```

ISA	Static Code Size	Inst. Count	Cycle Count	IPC
x86	45 (18)	80.7M	68.2M	1.18
Sparc	93 (55)	50.8M	47.2M	1.08
Alpha	100(44)	107.4M	72.4M	1.48

Alpha

```
s4addq $0,$3,$3
mulq $1,$4,$1
mulq $2,$5,$2
lda $4,20($30)
ldl $3,0($3)
addq $2,$1,$1
ldl $2,0($4)
mulq $3,$1,$1
lda $3,32($30)
addq $1,$2,$1
s4addq $0,$3,$0
stl $1,0($0)
ldl $0,0($11)
s4addq $0,$3,$0
```

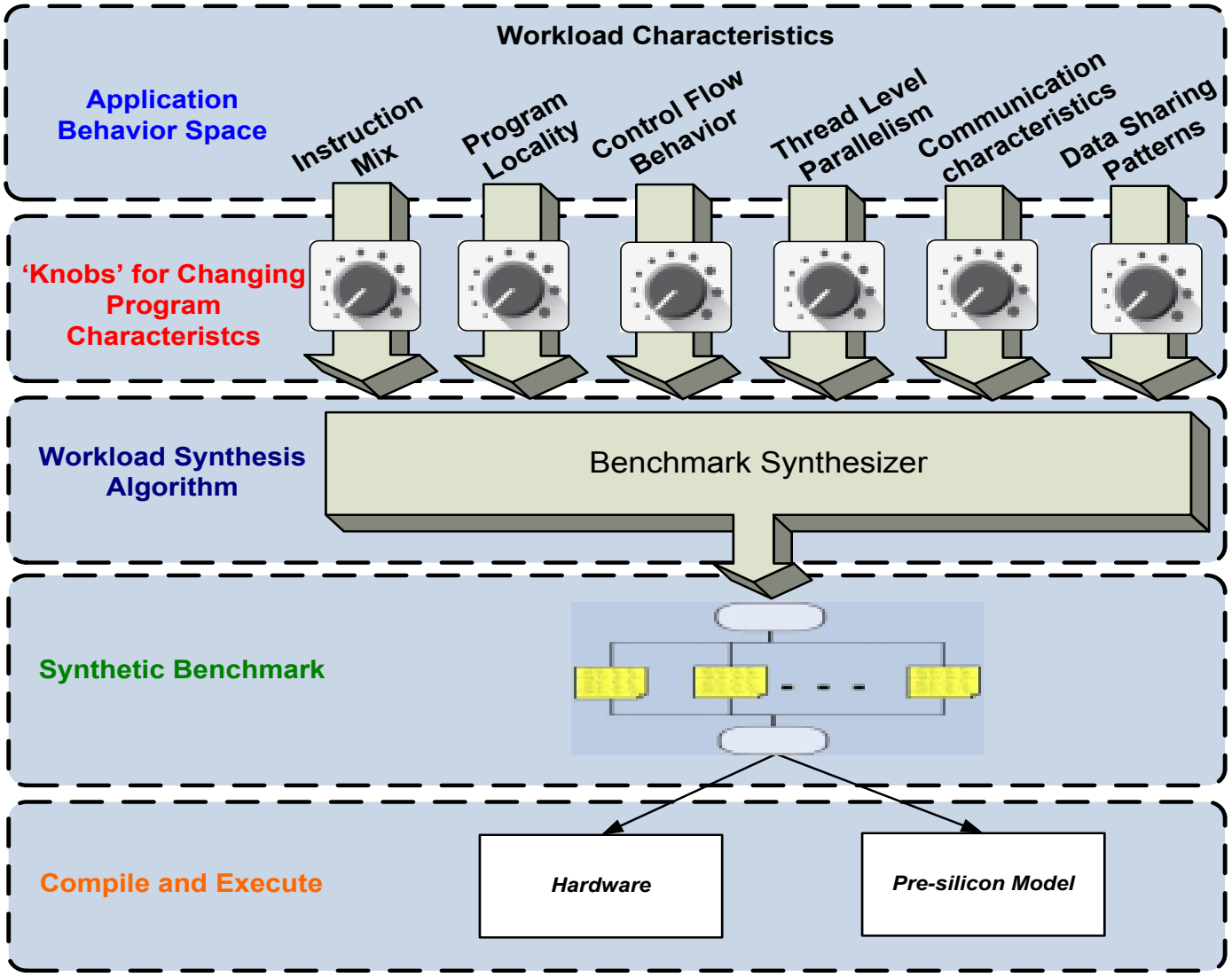
Sparc

```
sethi 4193522, %g1
add %g1, %i6, %g1
add %g1, 680, %l5
smul %l1, %l4, %l1
smul %l2, %l3, %l2
ld [%l5+%l0], %l3
add %l2, %l1, %l1
sethi 4193132, %g1
add %g1, %i6, %g1
ld [%g1+28], %l2
smul %l3, %l1, %l1
sethi 4193132, %g1
add %g1, %i6, %g1
add %g1, 40, %l3
add %l1, %l2, %l1
st %l1, [%l3+%l0]
sethi 4193132, %g1
add %g1, %l0, %g1
```

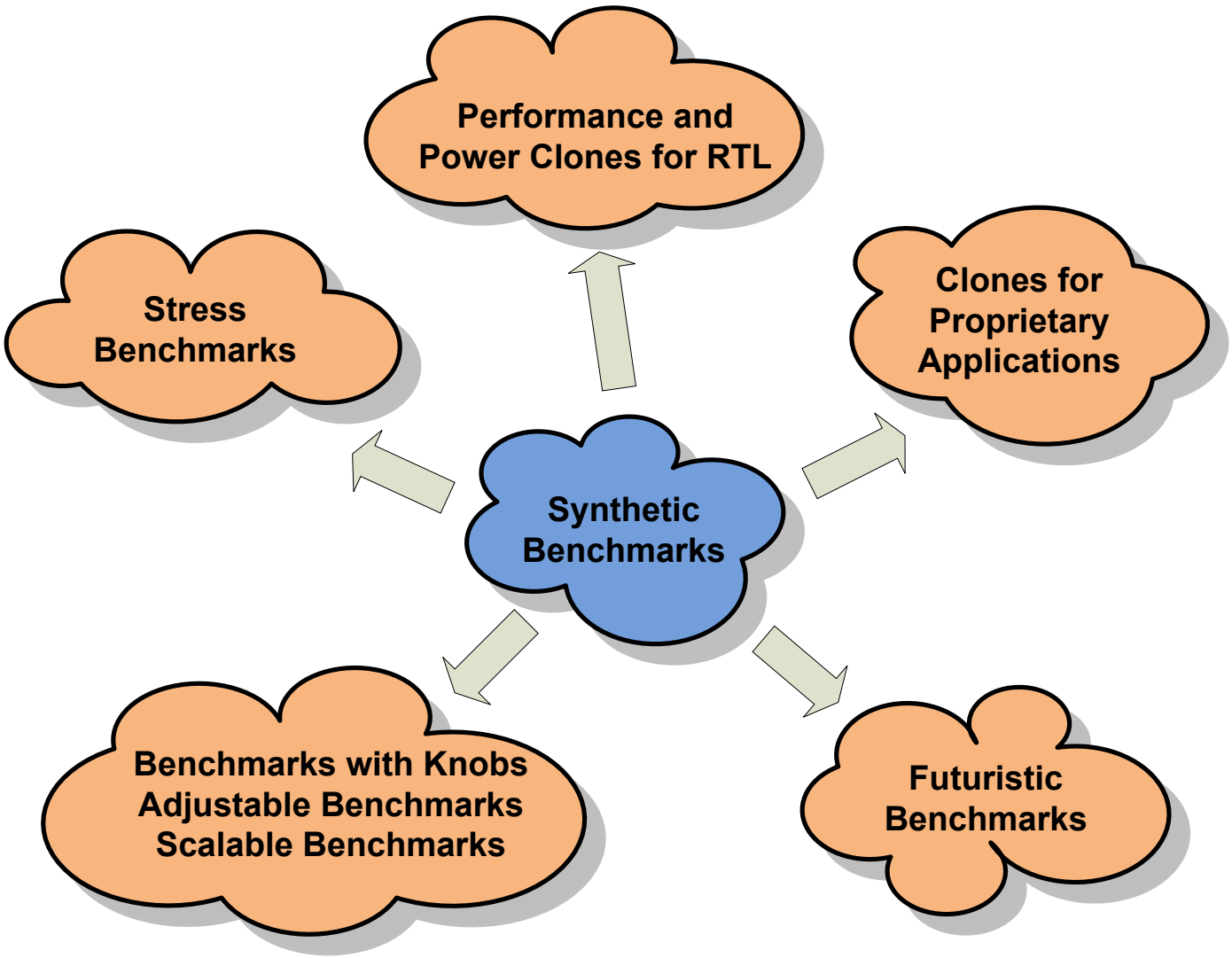
x86

```
movl    12(%esp), %eax
movl    800112(%esp,%eax,4), %ecx
imull   24(%esp), %ecx
movl    800108(%esp,%eax,4), %edx
imull   20(%esp), %edx
addl    %ecx, %edx
```

Other Applications

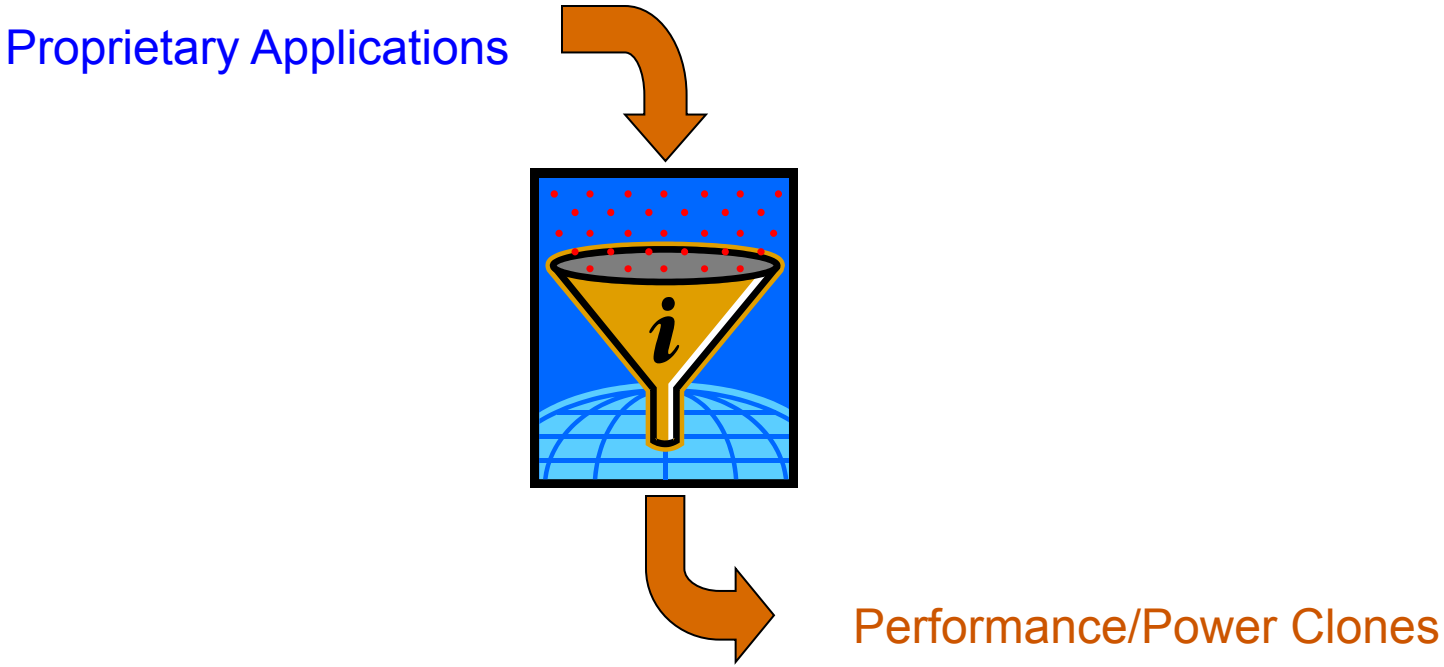


- Generate Clones by setting knobs to appropriate values
- Adaptable
- Scalable
- Futuristic


Performance and Power Clones for RTL
Stress Benchmarks
Clones for Proprietary Applications
Synthetic Benchmarks
Benchmarks with Knobs
Adjustable Benchmarks
Scalable Benchmarks
Futuristic Benchmarks

Sharing Proprietary Applications

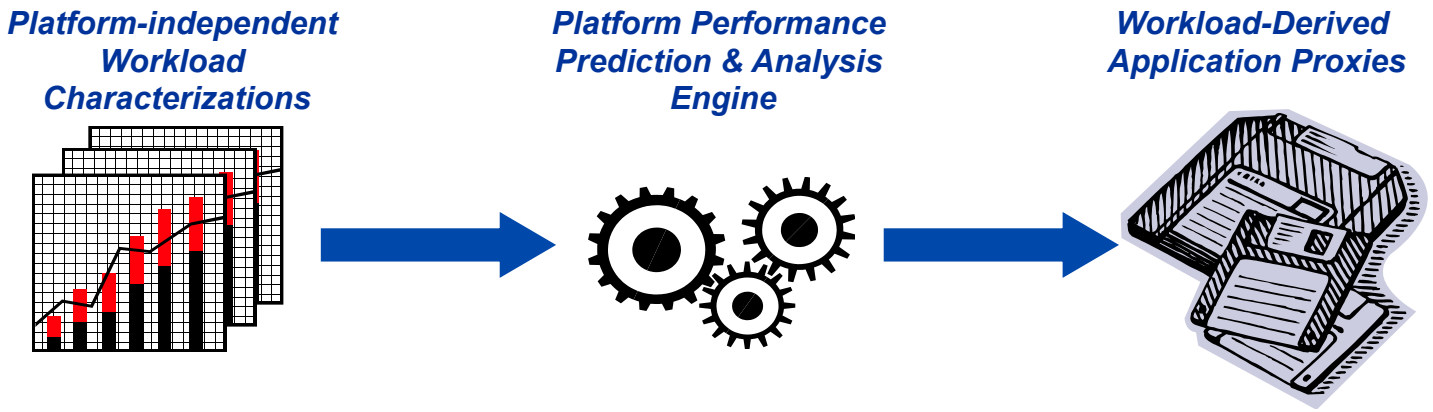
- Military Applications cannot be shared with Vendors
- Many commercial applications cannot be shared freely between software and hardware developers



Early Performance Testing

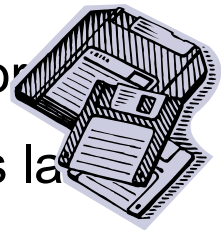
- **Usage Scenario**
  - Derive proxy applications from a set of workload characterizations
  - Proxies convey no proprietary information, but capture the execution behavior of XX's applications
  - Proxy applications can be available very early in project
  - Proxies focus on the computationally significant portions of code, not the small parts that are hardware-dependent (e.g., DMA, etc.)

➢ ***Early testing confirms performance, or discovers shortfalls in time to address them before final integration***



Proxy Applications can Transform Procurement Process

- **Proxy applications can turn vendors into partners in achieving performance goals**
  - Provides them with means to directly measure perfor
  - Places the iterative test/analyze process in vendor's la
  - Reveals no proprietary information



- **Turn hardware performance tuning into a supplier task**
  - Periodically capture application performance in a new set of proxies
  - Task supplier to re-tune hardware for best performance as applications mature

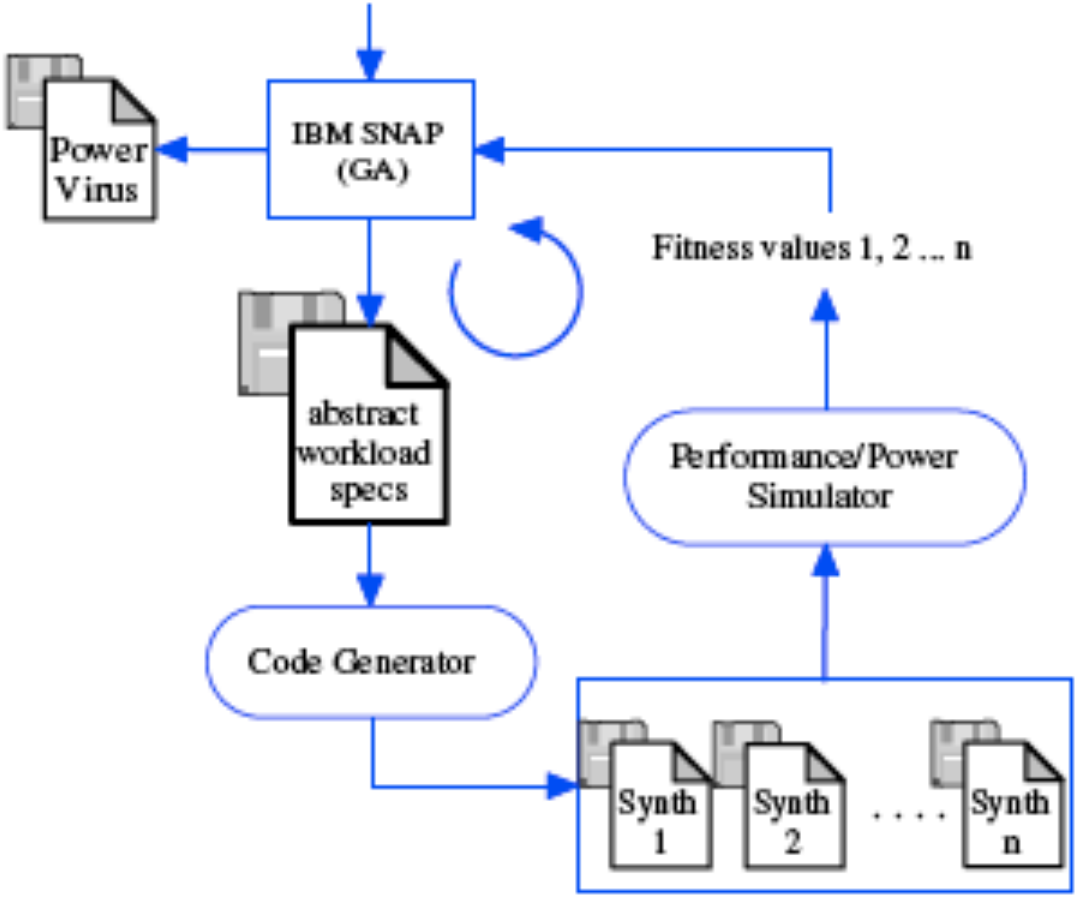
POWER AMD THERMAL STRESSMARKS

- **Hand crafting code snippets for power viruses**
  - Very tedious process, complex interactions inside the processor
  - Cannot be sure if it is the maximum case
  - Heavily architecture dependent; heavy domain knowledge
- **We automatically generate power viruses**

Power Virus	Optimized to stress	Language
MPrime	All CPUs, all ISAs	C
CPUburn-in	X86 machines	x86 assembly
BurnP5	Intel Pentium w&w/o MMX processors	x86 assembly
BurnP6	Intel PentiumPro, Pentium II, Pentium III and Celeron CPUs	x86 assembly
BurnK6	AMD K6 processors	x86 assembly
BurnK7	AMD Athlon/Duron processors	x86 assembly
BurnMMX	cache/memory interfaces on all CPUs with MMX	x86 assembly

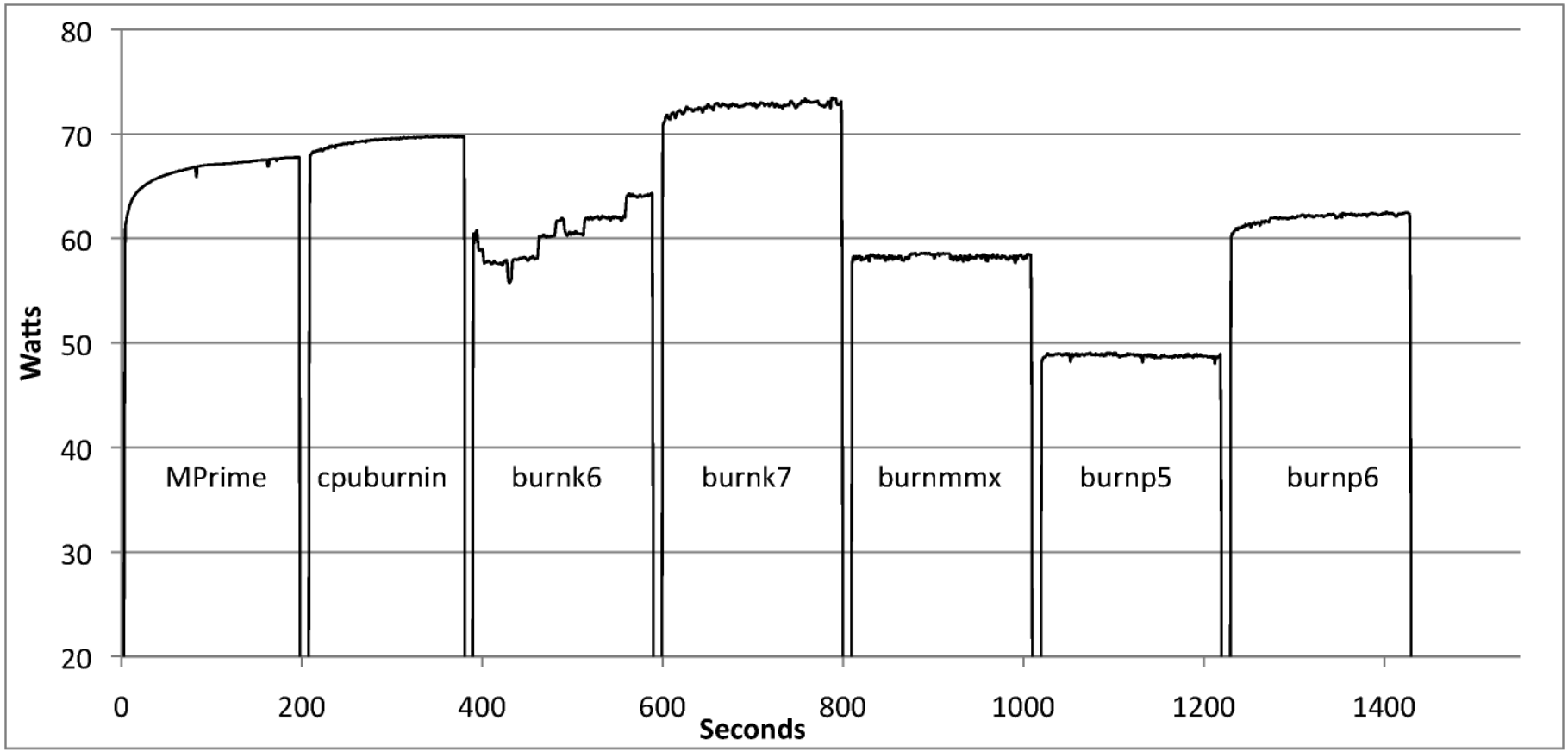
POWER and THERMAL STRESSMARKS

- **Automatically search for power viruses using an abstract workload model and machine learning**



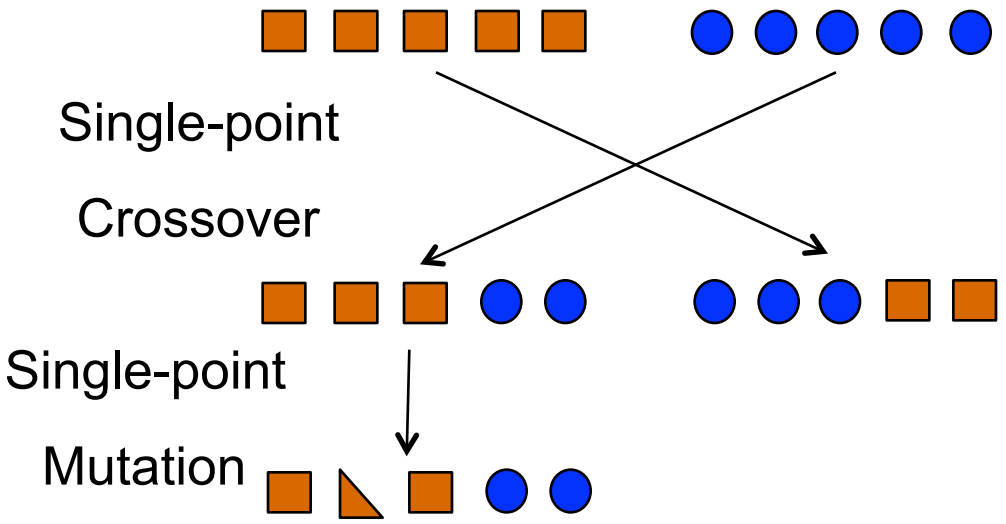
- **GA: search heuristic to solve optimization problems**
  - **Choose a random population, evaluate fitness, apply GA operators to generate next population**
- **Evolve until required fitness achieved**

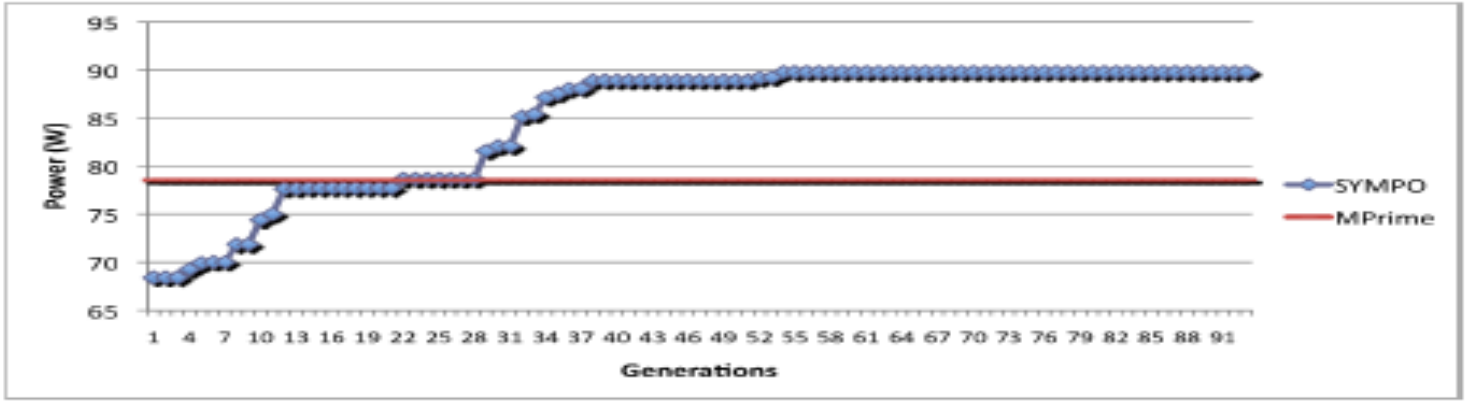
Power measurement of Viruses on Hardware



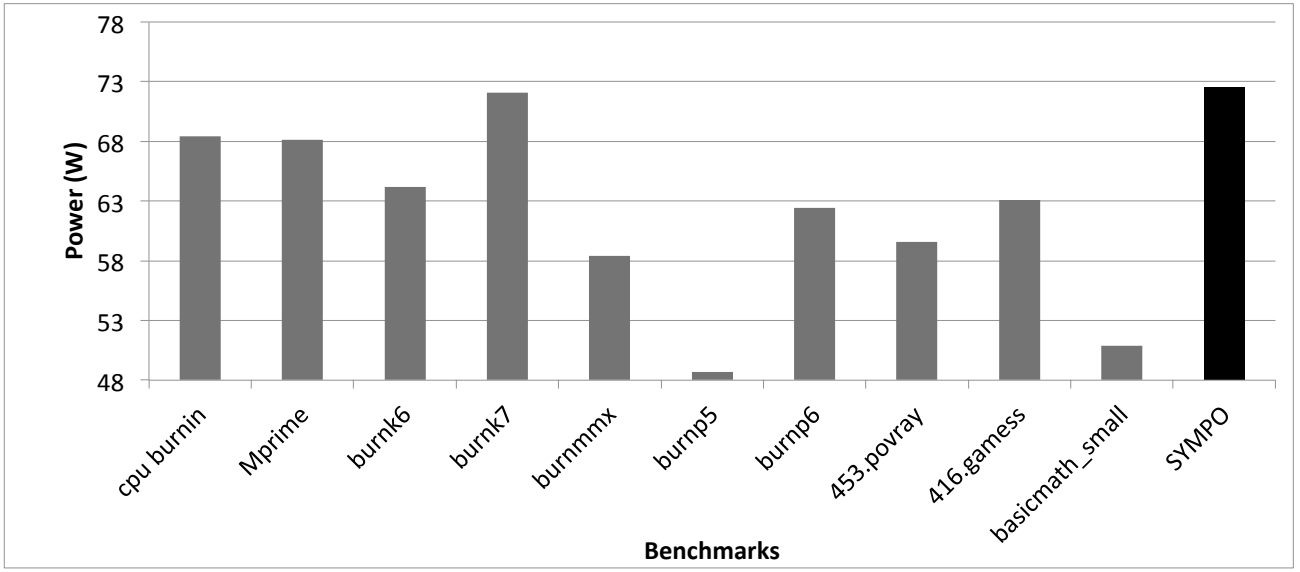
- **BurnK7 – 72.1 Watts**
- **SPEC CPU2006: 416.gamess and 453.povray consume highest power of 63.1 and 59.6 Watts**

SYMPO Framework – Genetic Algorithm





Validation on Real Hardware



- **Our code generator was not equipped to generate code for x86 at that time**
  - Generated power viruses on SPARC ISA and translated to x86 using LLVM infrastructure

A Solution to The Expiring Benchmark Problem

- SPEC89, SPEC92, SPEC95, SPEC CPU2000, CPU 2006
- TPC A, TPC B, TPC D

- Benchmarks become obsolete very quickly
- Benchmark makers create inflated data sets to stay alive at least 4-5 years
- Generate futuristic benchmarks using the proxy code generator

Interactive Workloads??

High-level feature categories need to be modified for interactive properties

1	Instruction-level Characteristics
2	Control-flow Characteristics
3	Memory-access Characteristics
4	Interactive Workload Characteristics

Weaknesses – The Vitamin Tablet Problem

Vitamin A, B, C, D, E, K

What about undiscovered vitamins? Vitamin P, Q, R?

Summary

- We have developed a proxy generation methodology to generate miniature proxy benchmarks.
- PerfProx proxies can enable fast and efficient performance evaluation of emerging workloads without needing back-end database or complex software stack support.
- We would like the community to use these proxies for architecture research to improve the reproducibility of the work with minimal effort.
- Automating the process will be helpful for deployment of our proxy generation tools (Intel project objective).

References

- ISCA 2004 - "Control Flow Modeling in Statistical Simulation for Accurate and Efficient Processor Design Studies" , Lieven Eeckhout, Robert Bell Jr., Bastiaan Stougie, Koen De Bosschere, Lizy K. John pdf
- ICS 2005 - "Improved Automatic Test case Synthesis for Performance Model Validation", Robert H. Bell, Jr. and Lizy K. John .
- IISWC 2005 - "Efficient Power Analysis using Synthetic Testcases", Robert H. Bell and Lizy K. John pdf
- IISWC 2006 - "Performance Cloning: A Technique for Disseminating Proprietary Applications as Benchmarks", Ajay Joshi, Lieven Eeckhout, Robert H.Bell Jr., and Lizy K. John pdf
- [ISPASS2006] Robert H. Bell, Rajiv R. Bhatia, Lizy John, Jeff Stuecheli, Ravel Thai, John Griswell, Paul Tu, Louis Capps, Anton Blanchard, "Automatic Testcase Synthesis and Performance Model Validation for High-Performance PowerPC Processors", Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS), March 2006, pp. 154-165.

References

- [HPCA2008] A. Joshi, L. Eeckhout, L. K. John, and C. Isen, "Automated Microprocessor Stressmark Generation", Proceedings of the IEEE International High Performance Computer Architecture (HPCA) Symposium, 2008, pp. 229-239.
- [ISPASS2010] Karthik Ganesan, Jungho Jo, and Lizy K. John, " Synthesizing Memory-Level Parallelism Aware Miniature Clones for SPEC CPU2006 and ImplantBench Workloads", 2010 International Symposium on Performance Analysis of Systems and Software. ISPASS, March 2010.
- [TC 2014] Karthik Ganesan and Lizy K. John, Automatic Generation of Miniaturized Synthetic Proxies for Target Applications to Efficiently Design Multicore Processors, IEEE Transactions on Computers, Vol. 63, No. 4, pp. 833-846, April 2014

- [PACT 2017] Reena Panda and Lizy K. John, Proxy Benchmarks for Emerging Workloads, International Conference on Parallel Architectures and Compilation Techniques (PACT), September 2017

References

- **[SAMOS] Reena Panda, Xinnian Zheng, Jee Ho Ryoo, Michael LeBeane, Shuang Song, Andreas Gerstlauer, and Lizy K. John, "Genesys: Automatically Generating Representative Training-sets", The IEEE International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS). July 2016**
- **[GLSVLSI17] S. Song, R. Desikan, M. Barakat, S. Sundaram, A. Gerstlauer, L. K. John, Fine-Grain Program Snippets generator for Mobile Core Design," Proceedings of the Great Lakes Symposium on VLSI, GLSVLSI, May 2017**
- **[ISPASS2017] Reena Panda, Xinnian Zheng, and Lizy K. John, " Accurate Address Streams for LLC and Byeond (SLAB): A Methodology to Enable System Exploration", International Symposium on Performance Analysis of Systems and Software (ISPASS), April 2017**
- **[DAC2017] Reena Panda, Xinnian Zheng, Jiajun Wang, Andreas Gerstlauer, and Lizy K. John, "Statistical Pattern Based Modeling of GPU Memory Access Streams", IEEE/ACM Design Automation Conference (DAC). June 2017.**
- [ICS2018] Panda, Reena, and Lizy K. John. "HALO: A Hierarchical Memory Access Locality Modeling Technique For Memory System Explorations." In Proceedings of the 2018 International Conference on Supercomputing, pp. 118-128. ACM, 2018.

References – Past Dissertations

- **Rob Bell, Ph. D Dissertation, UT Austin 2005, Automatic Workload Synthesis for Early Design Studies and Performance**
- **Ajay Joshi, Ph. D Dissertation, UT Austin2008, Constructing Adaptable and Scalable Synthetic Benchmarks for Microprocessor Performance Evaluation**
- **Karthik Ganesan, Ph. D dissertation, UT Austin 2012, Automatic Generation of Synthetic Workloads for Multicore Systems**
- **Reena Panda, Ph. Dissertation, UT Austin2017, Accurate Modeling of Core and Memory Locality for Proxy Generation Targeting Emerging Applications and Architectures**

The University of Texas at Austin

lca.ece.utexas.edu

BACK UP

Branch bit vectors

- Need a lightweight way to specify taken/transition rates of a branch
  - Minimize computational overhead to avoid skewing results
- Solution: Compare a 1-hot shift register against a static 32 bit vector
  - High transition rate branches use a register that shifts every iteration
  - Low transition rate branches use a register that shifts every 32 iterations
  - Branch if zero flag not set (both vectors had a 1 in the same position)
- Examples (using short vectors):
  - 11001100: 50% taken rate, 50% transition rate (1.6% with slow shift register)
  - 11101110: 75% taken rate, 50% transition rate
  - 10101010: 50% taken rate, 100% transition rate
  - 10000000: 12.5% taken rate, 25% transition rate

Generation Process

- Generate vector with target number of transitions
  - Roughly equal-sized groups of 0s and 1s, roughly 50% taken rate
- Flip bits to match target taken rate, without impacting transition rate
  - V[n] can be flipped if V[n-1] != V[n+1]
  - Example: 11110000 and 11100000 have the same transition rate
- Randomize the vector
  - Perform a random rotation
  - Shuffle groups of bits without impacting transition rate
    - Example: 11011010 and 11010110 have the same transition rate

COMPARISON WITH STANDARD BENCHMARKS