Machine Learning–Enabled Scalable Performance Prediction of Scientific Codes

Gopinath Chennupati *(now Amazon)*
Nandakishore Santhi
Phil Romero
Stephan Eidenbenz (eidenben@lanl.gov)

Los Alamos National Laboratory

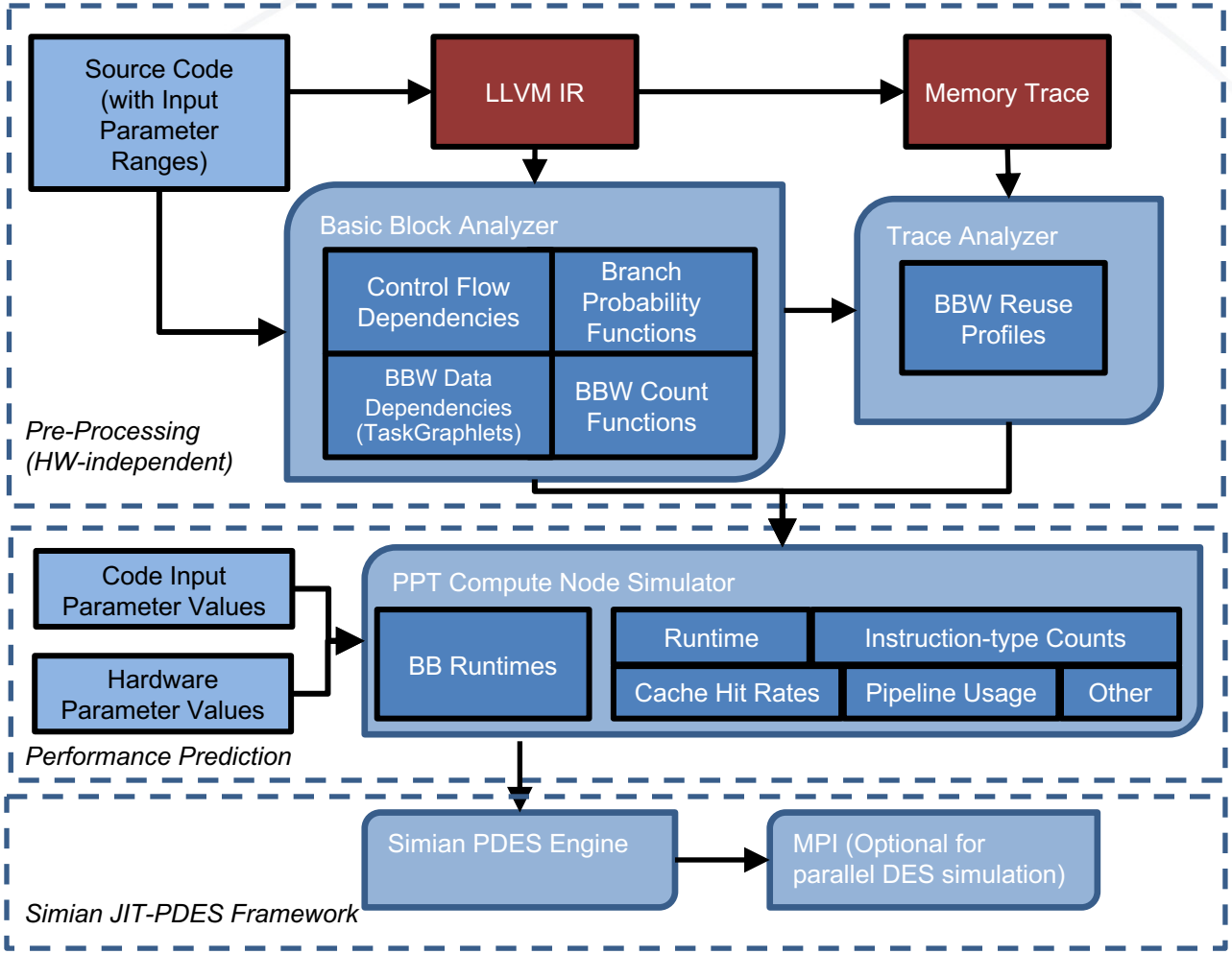
UNCLASSIFIED

Storyline

- **Goal:** fast and accurate runtime prediction and "bottleneckology" ("what is the limiting resource?") of scientific codes on modern hardware platforms
  - Incorporating effects of
    - Many/multi-cores, memory hierarchies, instruction pipelines
  - Restricted to single compute node
    - Coupling to MPI model in the works
- **Approach**
  - Learn functions of **basic block** execution counts, based on problem input parameters (eg matrix size)
  - Learn functions of how long an average, single execution of each basic block takes (including memory access and instruction pipelines)
  - Extrapolation: Learn based on small-scale examples, fit into closed-form formulas
- **Results**
  - Validation on 4 small benchmarks and 1 mini-app (SNAP)
  - Illustration of design-space exploration (e.g., vary cache size)

[Chennupati, G., Santhi, N., Romero, P., & Eidenbenz, S. (2021). Machine Learning–enabled Scalable Performance Prediction of Scientific Codes. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, *31*(2), 1-28.]

PPT-Analytical Memory Model with Pipelines



Legend:

Tool

Dataset

BBW = Basic Block – wise (for each basic block)

PPT = Performance Prediction Toolkit

Basic Block (BB):

- Linear sequence of instructions without branching
- Control flow dependency graph of any code with BB as nodes
- Introduced by compiler community

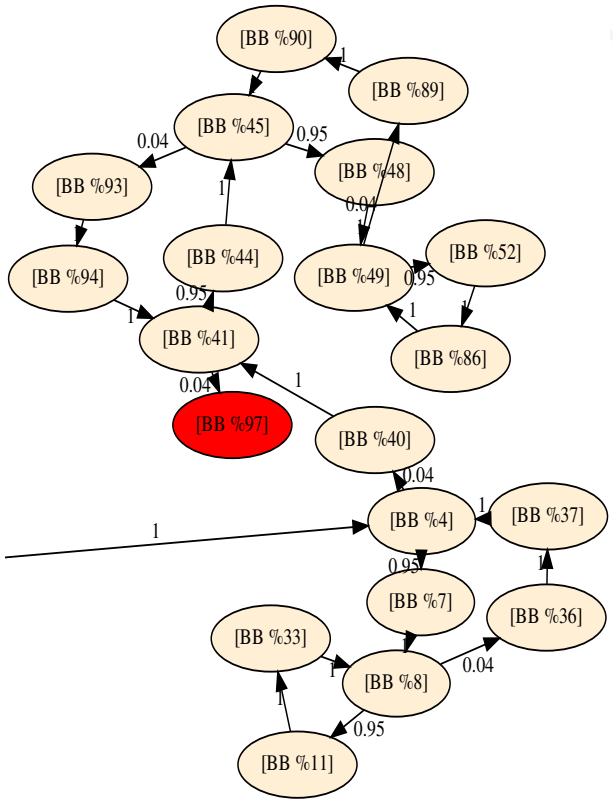
PPT: https://github.com/lanl/PPT

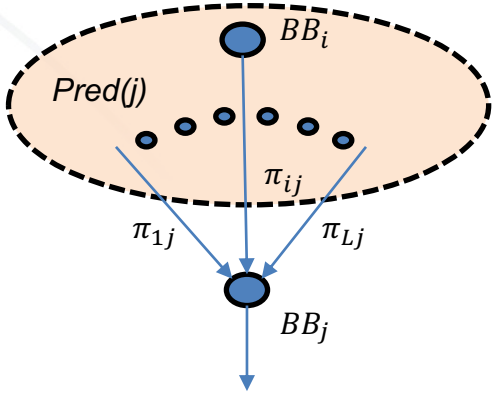
Simian PDES: https://github.com/pujyam/Simian

PPT-AMMP: *Basic Block Analyzer (BBA)*

```
#define N  256;
float ** r8_ ijk ( float ** a, float **
  b, float ** c)
{
  int i, j, k; // Initialization
  for (i =0; i <N; i++) // i loop
    for (j = 0; j < N; j++) { //
        j loop a[i][ j] =0.0;
        b[i][ j] = c[i][ j] =1.0;
      }
  for (i=0; i< N; i++) //
    i loop  for  (j=0; j<
    N; j++)  // j loop
      for (k =0; k< N; k++) // k loop
        a[i][ k] = a[i][ k] + b[i][ j]
  * c[j][ k];  return  a;
} // end of r8_ijk ()

int main () {
  float A[N][ N], B[N][ N], C[N][ N];
  A  = r8_ ijk
  (A,B,C);
  return 0;
} // end of main ()
```





$$P(BB_j) = \frac{N_j}{\sum_{k=0}^{n(BB)} N_k}$$

$$N_j = \sum_{i \epsilon Pred(j)} \pi_{ij} * N_i$$

$$N_1 = 1$$

TASK

1. Identify basic blocks (BB)
2. Build BB control flow dependency graph
3. Learn *branch probabilities* and *basic block execution counts* as functions of input size

APPROACH

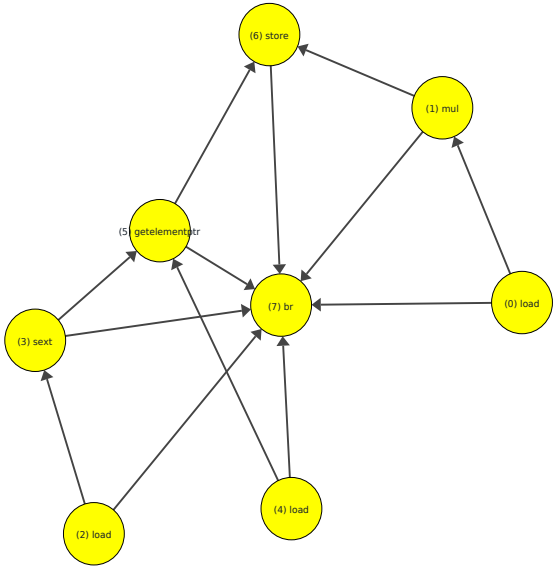
- Learn *closed-form formulas* based on training sets of small-input instances
- ML methods: Multi-linear regression, Genetic programming, CNN

PPT-AMMP: BBA Task graphlets for instruction-level parallelism

```
; <label>:6        ; preds = %3
%7 = load i32, i32* %i, align 4
%8 = mul nsw i32 2, %7
%9 = load i32, i32* %i, align 4
%10 = sext i32 %9 to i64
%11 = load i32*, i32** %1, align 8
%12 = getelementptr i32, i32* %11, i64 %10
store i32 %8, i32* %12, align 4
br label %13
```

Data Dependencies in a BasicBlock

Automated Data Dependency Graphlets



Task graphlet

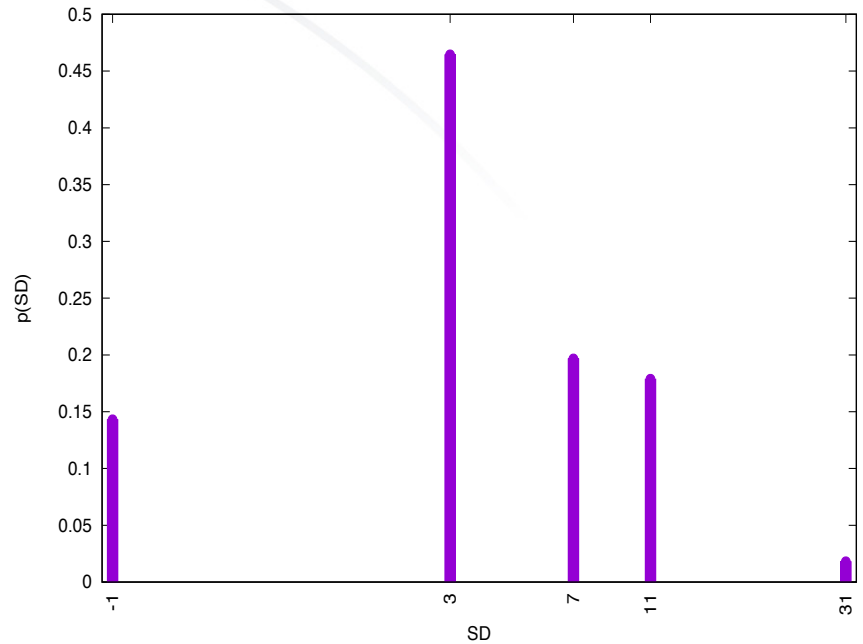
- **Goal:** Model instruction-level parallelism
- **Steps:**
  1. Build data dependency graph for each basic block (”task graphlet”), each node is a single instruction
  2. At Performance Prediction stage: Feed graph into HW pipelines model to predict execution runtime of an average execution of each basic block
- Key challenge: Load/Store (memory) operations … see next slide

PPT-AMMP: *Trace Analyzer*

BasickBlock Start: (main, entry)
LOAD: 0x7ffe5d1d8db0
BasickBlock End: (main, entry)
BasickBlock Start: (main, for_cond)
LOAD: 0x7ffe5d1d8dd0
STORE: 0x60410111ff0
BasickBlock End: (main, for_cond)
BasickBlock Start: (main, for_body)
LOAD: 0x7ffe5d1d8db3
STORE: 0x7ffe5d1d8db0
BasickBlock End: (main, for_body)

Reuse Distance (D)

Labeled Memory Trace



- **Goal:** Learn functions for average duration or cycle count of memory operations
- **Key concept:** Reuse distance (aka Stack distance)
  - Distribution of number of unique memory references between consecutive memory operations for the same memory cell
  - Independent of memory hierarchy, not easy to compute, O(n log n) with sampling
  - Well-known analytical formulas for cache hit rates based on reuse distance
  - Calculate average of distribution as a function of input parameters

[G. Chennupati, N. Santhi, S. Eidenbenz and S. Thulasidasan, *An analytical memory hierarchy model for performance prediction*, *2017 Winter Simulation Conference (WSC)*, Las Vegas, NV, 2017, IEEE. pp. 908-919.]

PPT-AMMP: *Compute Node Simulator*

Category	Parameter	Explanation
General	Clock speed	in Hz
Pipeline	Instruction Set	Group into instruction types, e.g: iALU, fALU, fDIV, mov, etc.
	Pipeline Counts	Number of pipelines per instruction type
	Latency	for each instruction pipeline
	Throughput	for each instruction pipeline
Memory	Cache Level Count	Number of cache levels
	Size	List of sizes for each cache level
	Latency	for each cache level
	Bandwidth	for each cache level
	Associativity	for each cache level
	Line Size	for each cache level
	RAM Size	
	RAM Latency	
	RAM Bandwidth	

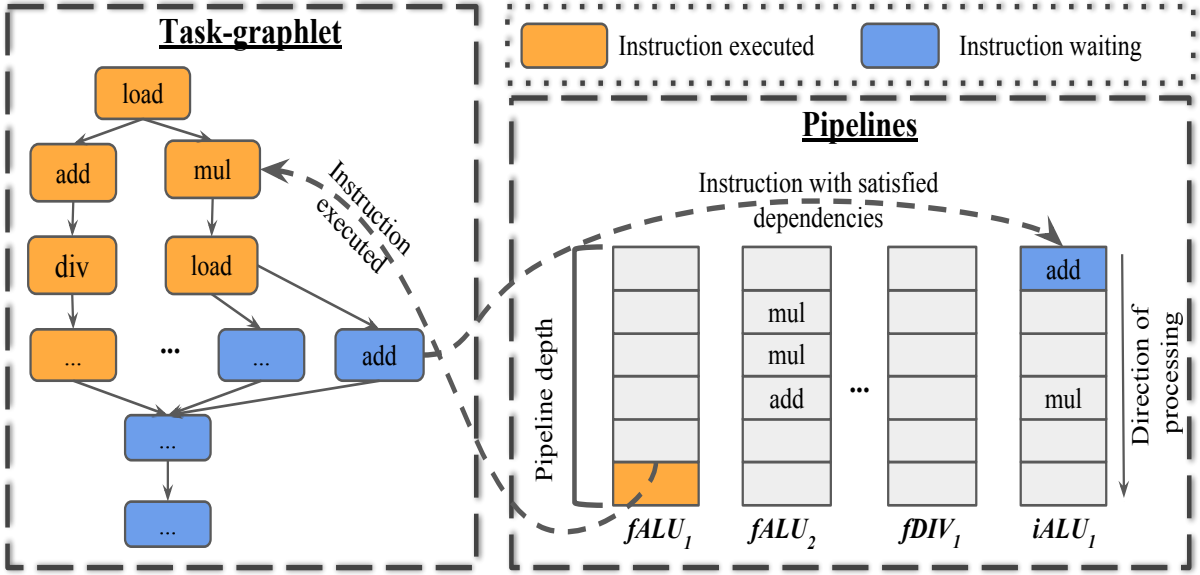
Hardware Parameters

Total runtime for all the basic blocks of a program

$$T = \sum_{i=0}^{n(BB)} t_i * N_i$$

The runtime t_i for i^{th} basic block and N_i is the number times that has been executed

PPT-AMMP: *Pipeline Simulator*



- ***Simian inside Simian***
- Pipelines count, depth
- Exercise for each *Basic Block* of the program

```
pipeline_ latencies ={ #in seconds
  'iadd' : 1.04e-9, 'fadd': 1.3e-9, '
  idiv': 9.46e-9, '
    fdiv': 15.46e-9, 'imul': 1.54e-9,
    'fmul': 2.31e-9,
    'load': 0.38e-9, 'store': 0.38e-9
     , 'alu': 0.38e-9, 'br': 0.38e-9,
     'unknown': 0.38e-9}

pipeline_ throughputs ={ #in seconds
  'iadd' : 0.25e-9, 'fadd': 0.38e-9, '
  idiv': 3.46e-9,
    'fdiv': 3.07e-9, 'imul': 0.5e-9, '
    fmul': 0.36e-9,'
    load': 0.38e-9, 'store': 0.38e-9,
    'alu': 0.38e-9,
    'br': 0.38e-9, 'unknown': 0.38e-9}
```

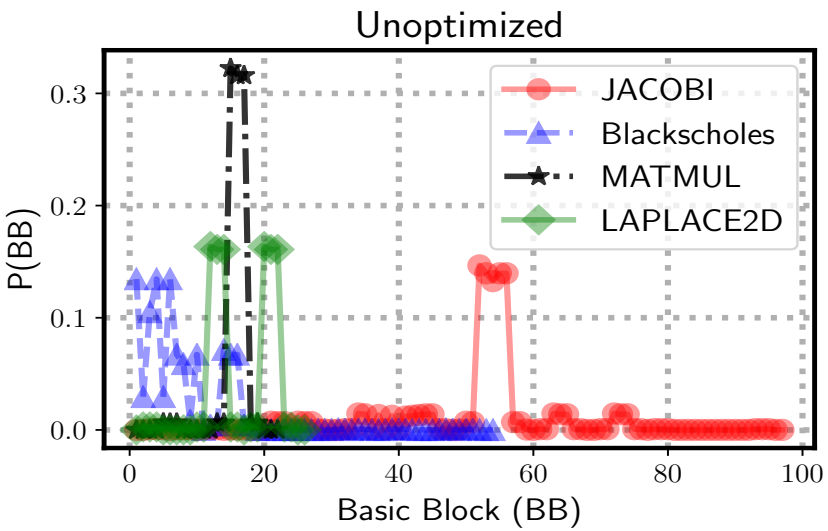
- **Goal:** Find average runtime for execution of a basic block
- **Approach:** Simulate HW pipelines on instruction-level
  - Classic queueing-style discrete event simulation (Simian engine)
  - Pipeline through-puts and latencies from published micro-benchmark results
  - Treat memory access as a special pipeline with average through-put and latency as computed by reuse-distance approach

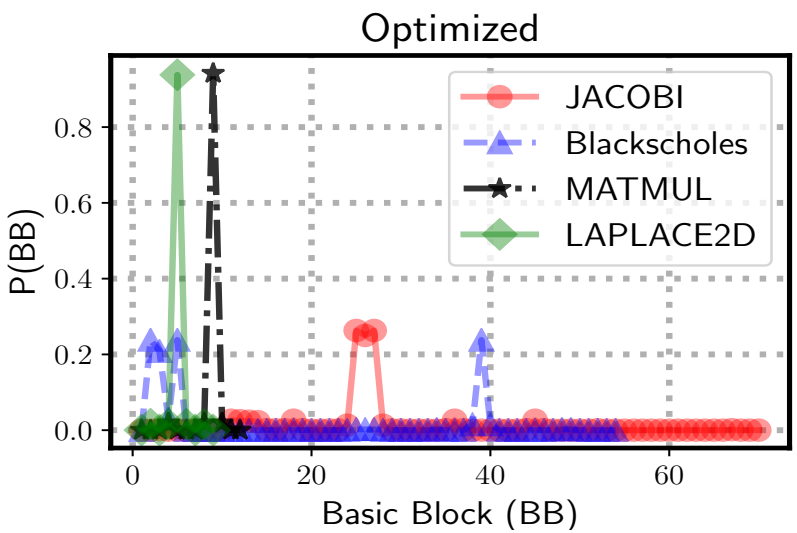
Experiments & Results

Benchmark	# program blocks		# kernel blocks	
	Unopt	Opt	Unopt	Opt
JACOBI	97	70	28	9
MATMUL	22	12	8	4
LAPLACE2D	26	9	6	1
BlackScholes	54	44	17	8

Number of Basic Blocks in a program and its kernel

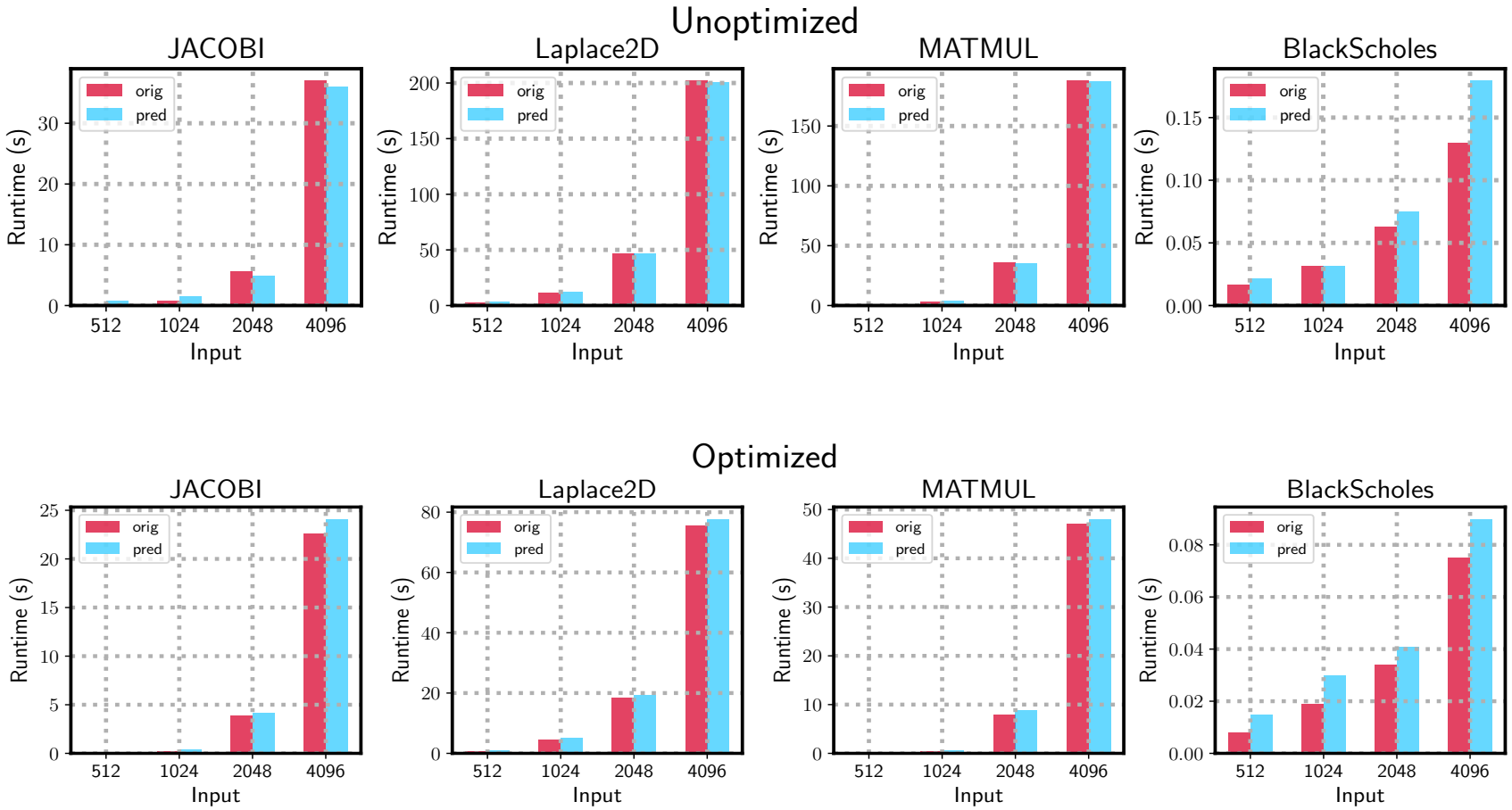
- Kernel blocks are the ones that have most significant influence on the program
- Probability distributions of various basic blocks
- *The higher the peak, the more significant is the basic block … only very few significant BBs per application*



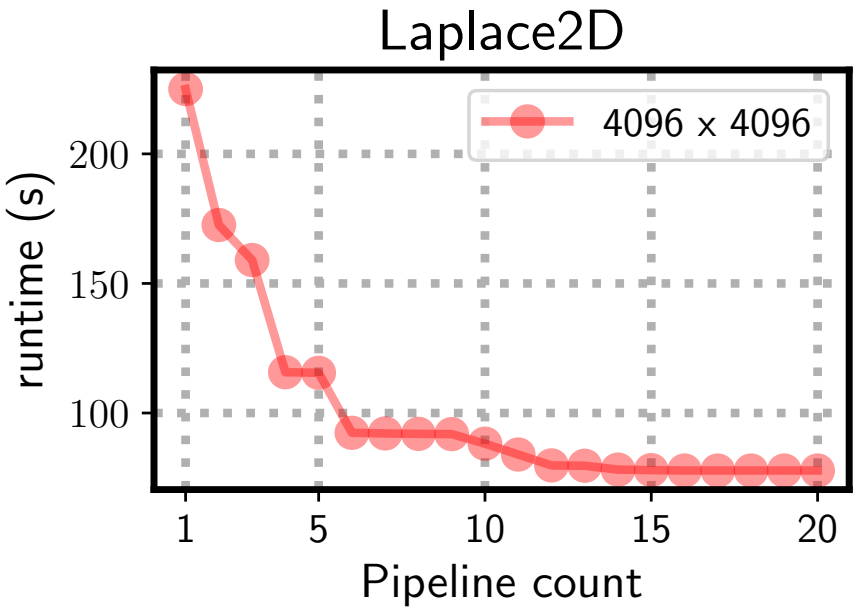


Discrete distributions of basic blocks with and without optimizations

Validating Compute Node Simulator

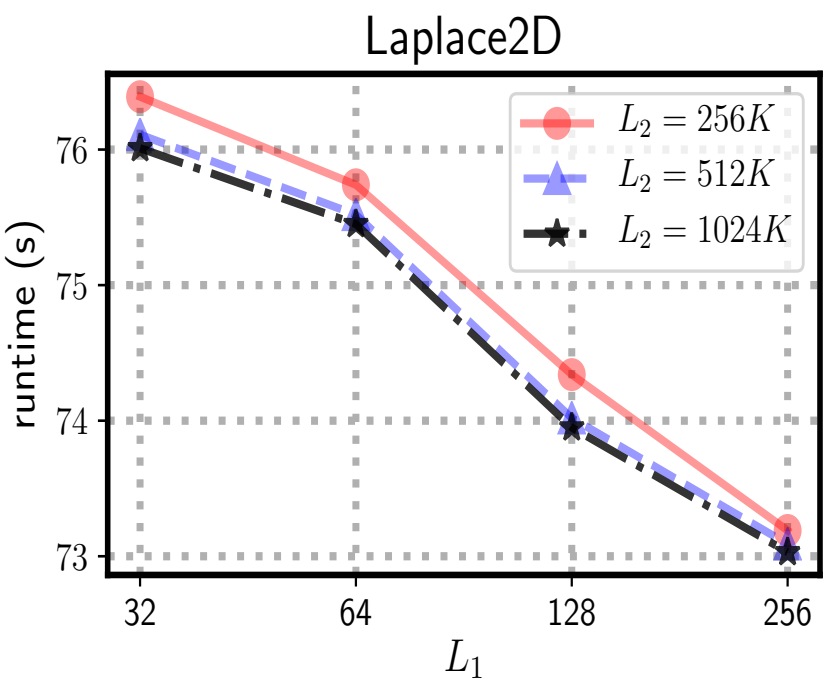


Sensitivity Analysis: Effect of Pipelines



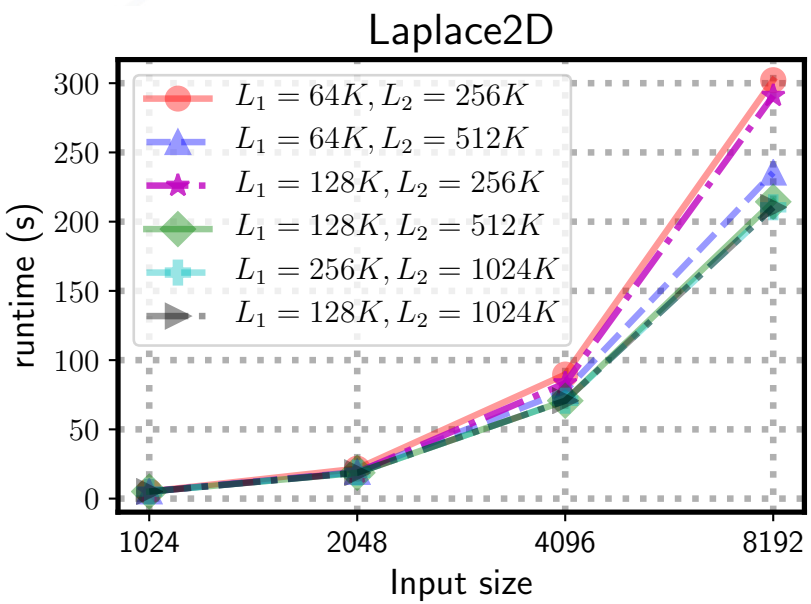
- Increase the number of pipelines for each of the CPU instructions
  - *iadd, fmul, fdiv, etc*
- Larger runtime at smaller pipeline count
- Runtimes *decrease* with the pipeline count and then *saturate*
- After a certain pipeline count, we observe *diminishing returns*

Sensitivity Analysis: Effect of Cache size



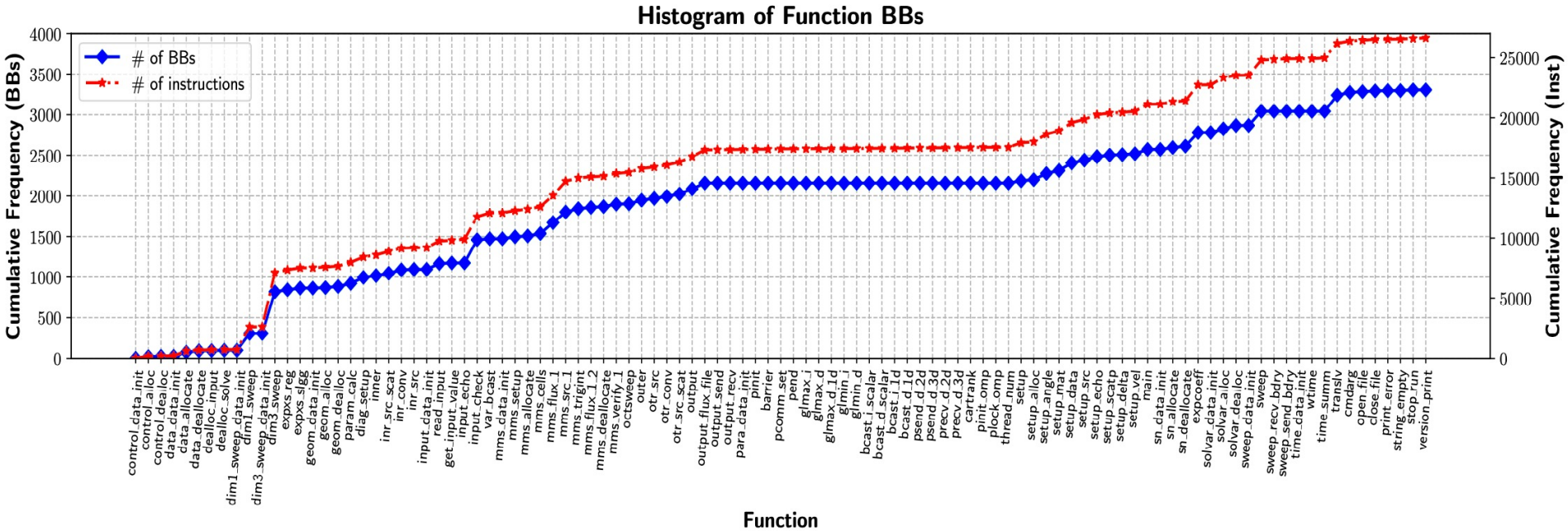
- Fixed input size. Vary L_1 and L_2 cache sizes
- Runtimes and cache sizes are inversely proportional
  - Runtime decreases with the increase in cache sizes
- For a smaller L_2 the runtime is expensive irrespective of L_1
- Relatively large L_1 and L_2 have positive impact on performance
- Further increase in L_2 results in diminishing returns - due to negative effects on latency, and smaller contiguous memory blocks meaning less bandwidth benefits

Sensitivity Analysis: Effect of Input Size on Cache



- Vary input work load and cache sizes (L_1 and L_2)
- Smaller input work load has insignificant effect on performance when cache sizes change
- At larger input work loads, cache sizes have significant impact on runtime
- Again, increasing cache sizes up to a certain (application dependent) threshold has positive impact on runtimes

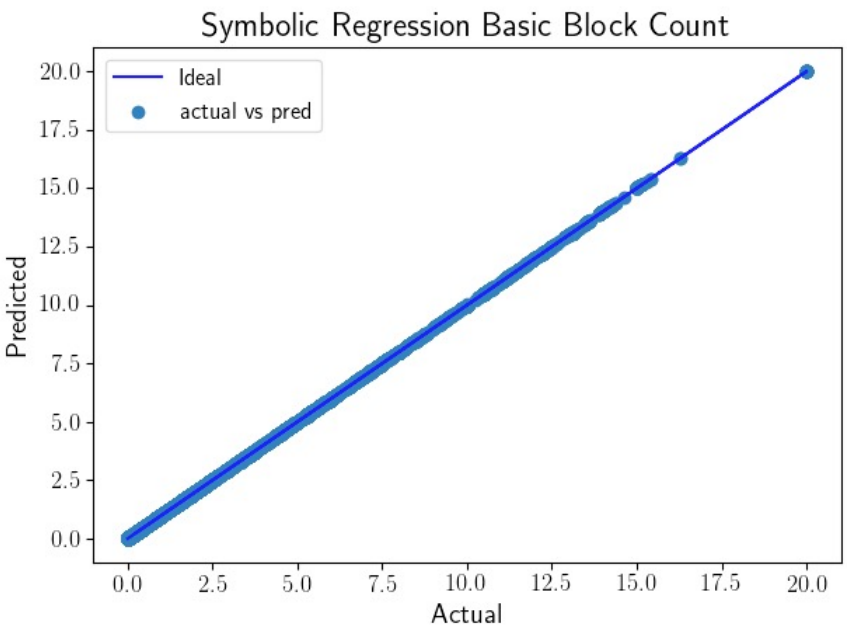
Mini-App SNAP results



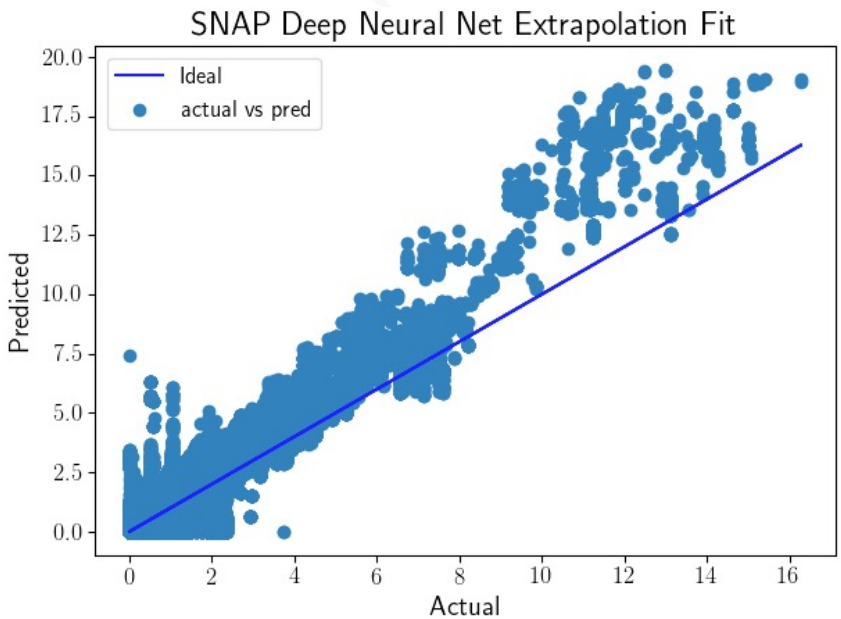
- **SNAP:** Discrete Ordinate Radiative Transport (“Sweep3D”)
- 100 x larger code than other benchmarks
- One-time Pre-processing and learning takes significant compute cycles (”days”)
- Performance prediction on any hardware and instance size takes seconds

SNAP: Learning BB count functions

Genetic programming ML



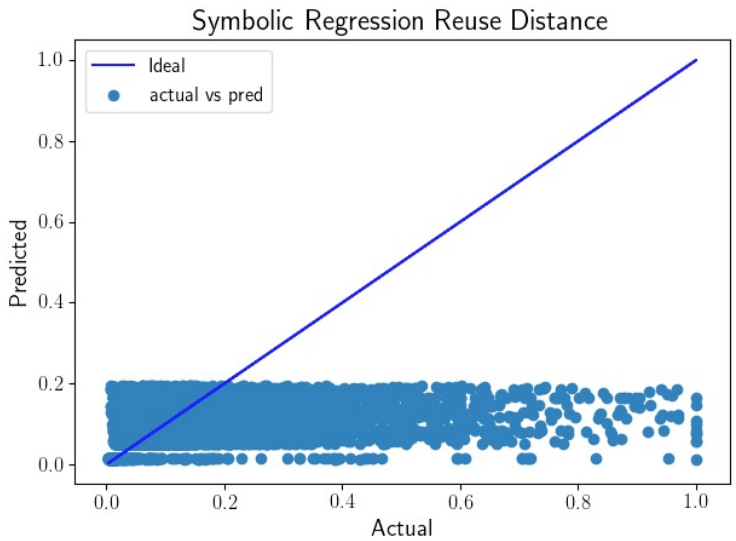
CNN Deep Learning



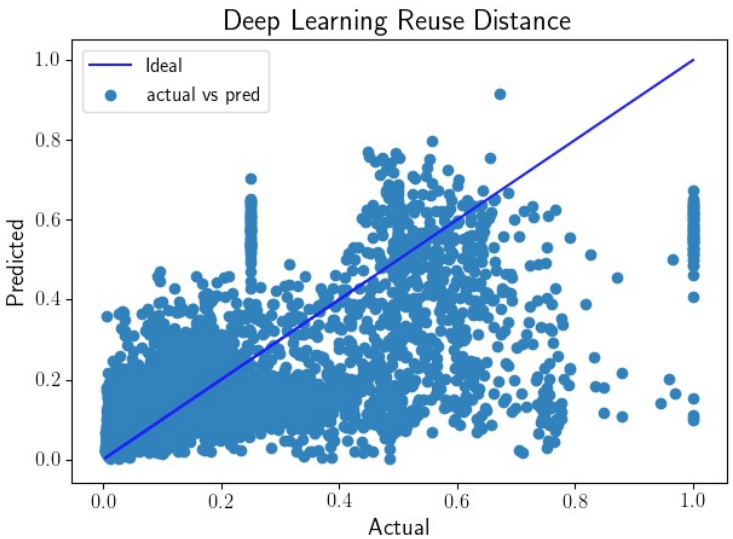
- Scatter plots Actual vs Predicted
- 4.8M (sampled) dots represent 6480 input combinations x 741 active basic blocks
- **Symbolic regression (Genetic programming based) outperforms CNN**

SNAP: Learning Reuse distance functions

Genetic programming ML



CNN Deep Learning



- Reuse distance learning is challenging for large codes
- **Deep learning outperforms Genetic programming**
- **Let's see if we still get good runtime predictions …**

SNAP: Overall runtime predictions on different input sets and platforms

#	Input									converged?	Runtime (s)	
	Nx	Ny	Nz	Ichunk	Nmom	Nang	Ng	Li	Lo		Actual	Predicted
in1	32	40	48	1	4	80	5	5	100	yes	51.58	58.02
in2	32	20	48	1	4	80	5	5	100	yes	151.92	156.74
in3	32	48	20	1	4	2	1	6	7	yes	5.012	5.901
in4	10	20	48	10	3	2	4	6	500	yes	20.66	20.31

Xeon E5-2695

Hardware	Parameters cache, sizes, cache line size, associativity, cache & RAM latency, threads, clock speed (Hz)	Input	Runtime (s)	
			Actual (±std)	Predicted
Ivy Bridge	3-level cache, (64 KB, 256 KB, 20 MB), (64, 64, 64), (8, 8, 12), (4, 12, 30) & 36.72 cycles, 16 threads/8 cores, 2.60 GHz `Intel E5-2650`	in1	9.164 (±0.91)	8.99
		in2	139.91 (±1.08)	145.53
		in3	2.86 (±0.74)	2.15
		in4	17.50 (±2.14)	18.01
Haswell	3-level cache, (32 KB, 256 KB, 25 MB), (64, 64, 64), (8, 8, 16), (4, 12, 43) & 62 cycles, 40 threads/20 cores, 3.10 GHz `Intel E5-2687W`	in1	7.318 (±1.11)	6.92
		in2	136.54 (±2.58)	143.38
		in3	2.27 (±1.68)	2.15
		in4	14.64 (±0.75)	15.33
Broadwell	3-level cache, (64 KB, 256 KB, 20 MB), (64, 64, 64), (8, 8, 20), 4, 12, 65) & 38 cycles, 64 threads/32 cores, 2.10 GHz `Intel E5-2683`	in1	8.215 (±0.62)	7.45
		in2	139.88 (±1.89)	142.43
		in3	2.46 (±2.05)	2.13
		in4	16.72 (±1.25)	15.88
Skylake	3-level cache, (64 KB, 1 MB, 19.25 MB), (64, 64, 64), (8, 8, 20), (4, 14, 38) & 42 cycles, 24 threads/12 cores, 3.40 GHz `Intel Gold 6138`	in1	6.812 (±1.10)	7.16
		in2	63.31 (±2.88)	67.10
		in3	2.03 (±0.54)	1.85
		in4	13.33 (±1.01)	14.68

Prediction errors generally less than 10%

Conclusions & Future

- PPT-AMMP is faster, scalable and relatively accurate
- We modeled the hardware memory/cache hierarchy and instruction pipelines
- Studied effect of pipelines, cache and input sizes on application performance for various benchmark codes
- Future: employ these techniques on more large scale apps
- Couple the PPT-AMMP models to our existing MPI interconnect models in PPT
- Recent extension covers GPU functionality (ACM SC2021 paper)

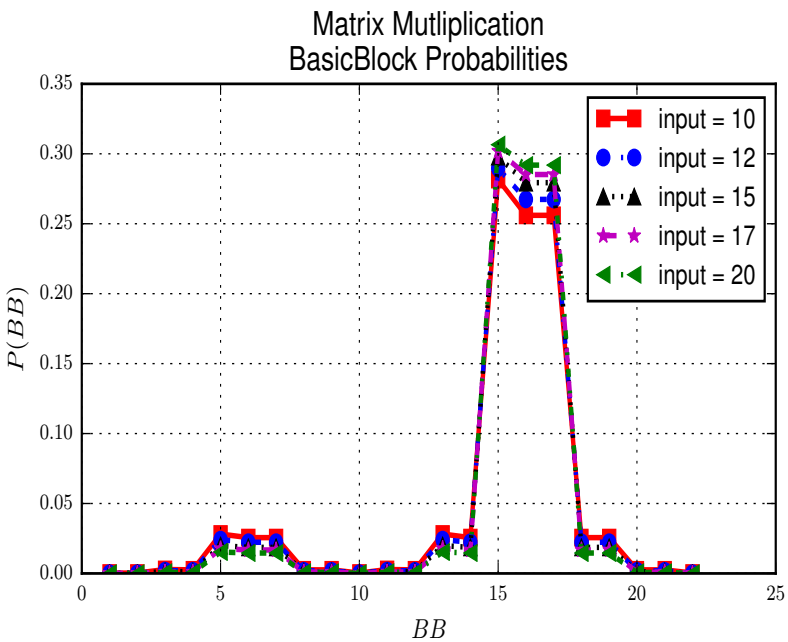
References

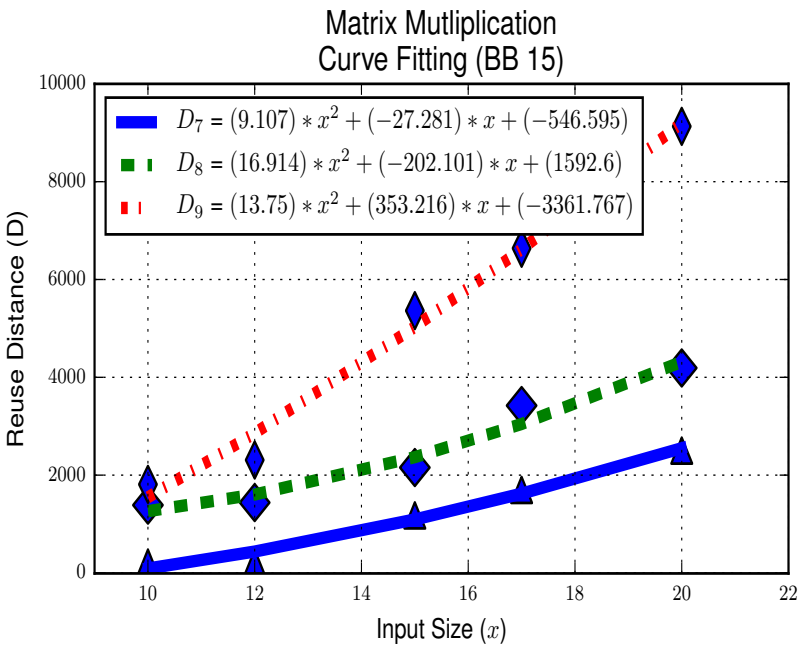
- M. Wu and D. Yeung, "*Efficient Reuse Distance Analysis of Multicore Scaling for Loop-Based Parallel Programs*", ACM Transactions on Computer Systems (TOCS); Volume 31 Issue 1, February 2013. Article No. 1
- N. Santhi, S. Eidenbenz, and J. Liu, "The Simian concept: Parallel Discrete Event Simulation with interpreted languages and just-in-time compilation", 2015 Winter Simulation Conference (WSC)
- C Hannon, D Jin, N Santhi, S Eidenbenz, J Liu. "Just-in-time parallel simulation", 2018 Winter Simulation Conference (WSC), 640-651
- G. Chapuis, S. Eidenbenz, and N. Santhi, "Gpu performance prediction through parallel discrete event simulation and common sense," in Proceedings of the 9th EAI International Conference on Performance Evaluation Methodologies and Tools, VALUETOOLS'15, (ICST, Brussels, Belgium, Belgium)
- M. Andersch, J. Lucas, M. Alvarez-Mesa, and B. Juurlink, "Analyzing gpgpu pipeline latency," in Proc. 10th Int. Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems, Fiuggi, Italy (ACACES' 14), July 2014.
- Hall, Piriya Kristofer, "Adaptation of a GPU simulator for modern architectures" (2016). Graduate Theses and Dissertations. 15712.
- Professional CUDA C Programming. Birmingham, UK, UK: Wrox Press Ltd., 1st ed., 2014.
- T.Tang, X.Yang, and Y.Lin. Cache Miss Analysis for GPU Programs Based on Stack Distance Profile. In ICDCS-31: Distributed Computing Systems. IEEE, 2011.
- Nugteren, Cedric and Van den Braak, Gert-Jan and Corporaal, Henk and Bal, Henri. A Detailed GPU Cache Model Based on Reuse Distance Theory. In Proceedings - International Symposium on High-Performance Computer Architecture
- Y. Arafa, AHM. Badawy, G. Chennupati, N. Santhi, S. Eidenbenz, PPT-GPU: Performance Prediction Toolkit for GPUs – Identifying the impact of caches, *International Symposium on Memory Systems (MEMSYS)*, Washington, USA, 2018, In Press.
- G. Chennupati, S. Eidenbenz, A. Long, O. Tkachenko, J. Zerr, and J. Liu, IMCSIM: Parameterized Performance Prediction For Implicit Monte Carlo Codes, *Winter Simulation Conference (WSC)*, Gotheburg, Sweden, 2018, In Press.
- B. Kalla, N. Santhi, A. A. Badawy, G. Chennupati and S. Eidenbenz, *Probabilistic Monte Carlo simulations for static branch prediction*, *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*, San Diego, CA, 2017, pp. 1-4.
- G. Chennupati, N. Santhi, R Bird, S. Thulasidasan, AHA Badawy, S Misra and S. Eidenbenz *A scalable analytical memory model for cpu performance prediction*, International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Systems (PMBS@SC), Denver, CO, USA. 2017
- G. Chennupati, N. Santhi, S. Eidenbenz and S. Thulasidasan, *An analytical memory hierarchy model for performance prediction*, *2017 Winter Simulation Conference (WSC)*, Las Vegas, NV, 2017, IEEE. pp. 908-919.
- G. Chennupati, N. Santhi, A. Eidenbenz,, R. J. Zerr, M. Rosa, R. J. Zamora, E-J. Park, B. T. Nadiga, J. Liu, K. Ahmed, and M. A. Obaida, *Performance Prediction Toolkit,* LANL, Los Alamos, NM, USA. 2017 *https://github.com/lanl/PPT*

BACKUPS

Extrapolating Reuse Profiles

- Reuse distances (D) does not grow linearly
- Fixed-Binning on fractions of reuse distance distribution: Learn functions for average reuse distances at each bin … then take average over those
- Can be done basic-block wise, but averaging over entire code works as well





[G. Chennupati, N. Santhi, R Bird, S. Thulasidasan, AHA Badawy, S Misra and S. Eidenbenz *A scalable analytical memory model for cpu performance prediction*, International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Systems at Super Computing (PMBS@SC), Denver, CO, USA. 2017]

PPT-AMMP: Hit-rate, Latency, *Runtime*

Reuse Profile : $$\Pr(D) = \sum_{i=0}^{n(BB)} P(BB_i) * P(D|BB_i)$$

Predicted runtime: $$T_{Pred} = \boldsymbol{T_{avg\_mem}} + \boldsymbol{T_{CPU\_ops}}$$

$$\boldsymbol{T_{avg\_mem}} = \frac{\lambda_{avg} + (\boldsymbol{b}-1)\,\beta_{avg}}{\boldsymbol{b}} * total\_mem$$

$\boldsymbol{b}$ is the average size of contiguous memory blocks accessed

Average Latency:

$$\lambda_{avg} = \boldsymbol{P_{L_1}(h)} * \lambda_{L_1} + \left(1 - \boldsymbol{P_{L_1}(h)}\right)\left[\boldsymbol{P_{L_2}(h)} * \lambda_{L_2} + \left(1 - \boldsymbol{P_{L_2}(h)}\right) * \lambda_{L_{RAM}}\right]$$

Average Reciprocal Throughput:

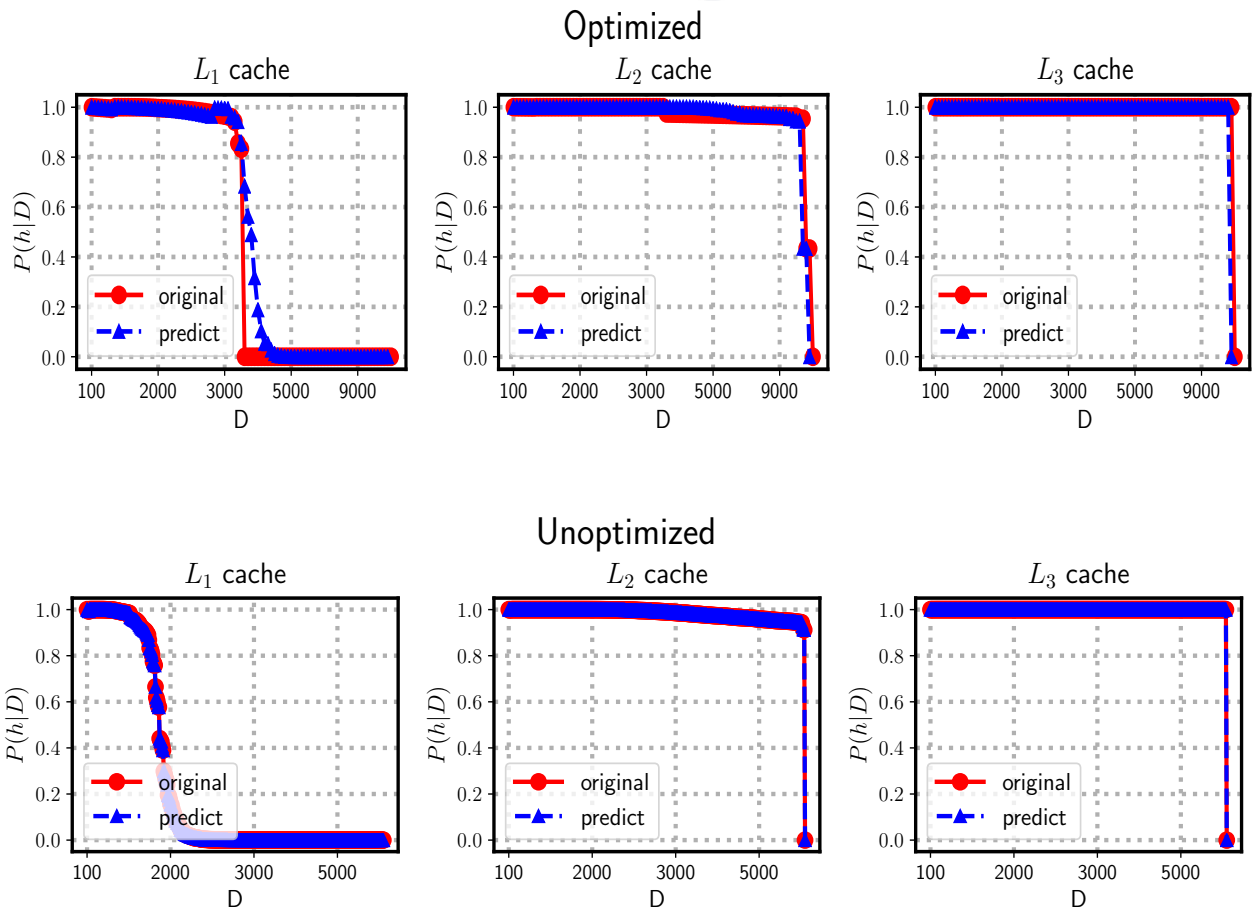
Similarly, measure β_{avg}

Cache hit-ratio model (for common cache architecture):

$$P(h|D) = \sum_{a=0}^{A} \binom{D}{a} \left(\frac{A}{B}\right)^{a} \left(\frac{B-A}{B}\right)^{(D-a)} \qquad P(h) = \sum_{i=0}^{N} P(D_i) * P(h\,|D_i)$$

Validating Trace Analyzer

- Final conditional hit-rates at a given reuse distance for a given cache (for JACOBI application, others similar)
- Predicted hit-rates match approximately with that of the real ones
- Useful for estimating the effective latencies and reciprocal throughputs



SNAP: Symbolic Regression Function examples

Function	Basic block	Equation
dim3_sweep	*for.inc115*	$1.72 \times Nx \times Ny \times Nz \times Ng \times Lo \times (Li + 0.23 \times \sqrt{Li})$
	vector.body646	$0.073 \times Nx \times Ny \times Nz \times Ng \times Li \times Lo \times (Nmom - 0.62/Nmom)$
	min.iters	$1.63 \times Nx \times Ny \times Nz \times Ng \times Li \times (Lo + 0.317 \times \sqrt{Lo})$
mms_flux_1	*vector.body4720*	$0.20 \times Nz \times Ng \times Lo \times (2.49 - 0.001 \times Lo \times \sqrt{Nz})$
	land.lhs	$0.21 \times Nz \times Ng \times Lo \times (2.38 - 0.0004 \times Nz \times Lo)$
	if.then272	$0.25 \times Ny \times Ng \times Lo \times (Nz + 0.086/Lo)$
translv	*for.cond354*	$0.221 \times Nx \times Ny \times Nz \times Ng \times Lo \times (Li + 0.184 \times \sqrt{Li})$
	for.inc425	$0.411 \times Ng \times Li \times Lo \times (2.452 - 0.004 \times Li \times Lo)$
	middle.block1195	$0.484 \times Nz \times Ng \times Lo \times (Li + (-0.204)/((Li - 3.854 \times Ng)))$
	if.end103	$0.0862 \times Nx \times Ny \times Nz \times Ng \times Li \times Lo \times (Nmom - 0.484/(Nmom))$

Existing Prediction Frameworks

Simulators – try to mimic the architectures (often cycle accurate)

- RSIM (1997) – only multi-core processor at that time
- SimpleScalar (2002) – complex branch predictions, instruction sets
- Gem5 (2011), McSimA+ (2013), Zsim (2013),
- Manifold (2014) – multi/many-core

Parallel Discreate Event based Simulator (PDES)

#	Simulator	Highlights
1	BigSim (2004)	MPI simulator, accurate, may not scale to large core counts
2	Extreme scale simulator (xSim, 2011)	Large scale interconnect simulator, simple test programs
3	Structural Simulation Toolkit (SST, 2011)	Simulates almost everything, not enough test cases
4	Co-design exascale storage system (CODES, 2011)	ROSS-based simulator, mostly accurate, and scalable. Ease-of-use?

More high level modeling:

- Aspen (ORNL, 2012), Durango (ANL, 2017), **Simian + PPT (LANL, 2017)**, etc.

Where We Stand -- Scalable Simulation?

#	Framework	Simulated	Physical Ranks
1	BigSim (2004)	64,000 processors	--
2	xSim (2010)	1.048 million ranks	--
3	SST (2011)	--	--
4	**CODES (2011)**	**1 billion ranks**	16,384 cores
5	**Simian + PPT (2017)**	16.5 million ranks	**4,096 ranks**

and we are nowhere near stressing the simulator yet …

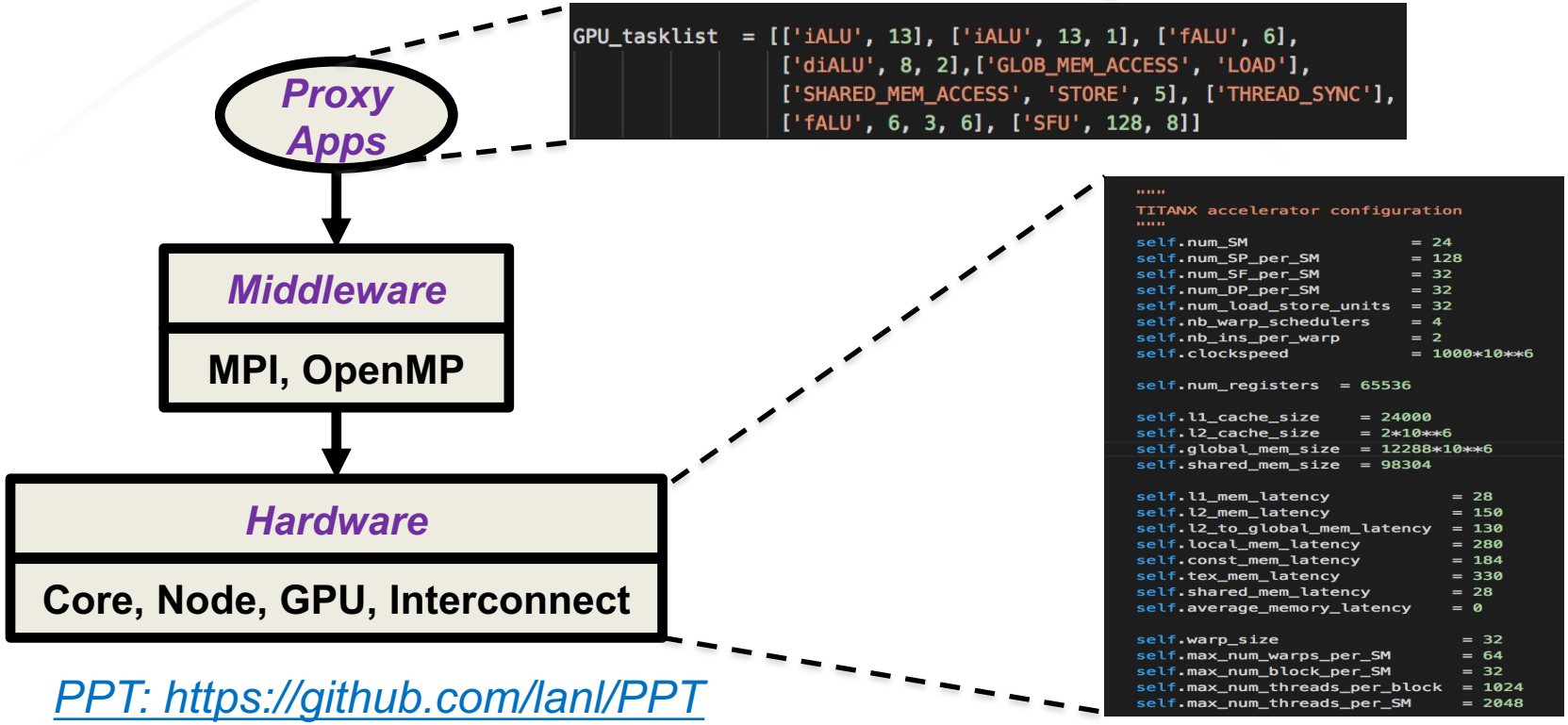
N. Santhi, J. Liu, S. Eidenbenz, The Simian Concept: Parallel Discrete Event Simulation With Interpreted Languages and Just-In Time Compilation, Winter Simulation Conference (WSC), Huntington Beach, USA, 2015.

K. Ahmed, M. Obaida, J. Liu, S. Eidenbenz, N. Santhi, G. Chapuis, An Integrated Network Model for Large-Scale Performance Prediction, ACM SIGSIM-PADS, 2016.

G. Chennupati, S. Eidenbenz, A. Long, O. Tkachenko, J. Zerr, and J. Liu, IMCSIM: Parameterized Performance Prediction For Implicit Monte Carlo Codes, *Winter Simulation Conference (WSC)*, Gotheburg, Sweden, 2018.

M. Obaida, J. Liu, G. Chennupati, N. Santhi, S. Eidenbenz, Parallel Application Performance Prediction Using Analysis Based Models and HPC Simulations. SIGSIM-PADS 2018: ACM. pp: 49-59

Performance Prediction Toolkit (PPT)



[PPT: https://github.com/lanl/PPT](https://github.com/lanl/PPT)
[Simian PDES: https://github.com/pujyam/Simian](https://github.com/pujyam/Simian)

G. Chennupati, N. Santhi, S. Eidenbenz, R. J. Zerr, M. Rosa, R. J. Zamora, E-J. Park, B. T. Nadiga, J. Liu, K. Ahmed, and M. A. Obaida. *Performance Prediction Toolkit*, LANL, Los Alamos, NM, USA. 2017