LA-UR-17-20754
Approved for public release; distribution is unlimited.

Title: Compute Node Models: Large-scale Amenable Block-Level Simulation for Memory Hierarchies and Pipelines

Author(s): Santhi, Nandakishore
Chennupati, Gopinath

Intended for: Report

Issued: 2017-02-01

Compute Node Models: Large-scale Amenable Block-Level Simulation for Memory Hierarchies and Pipelines

January 26, 2017

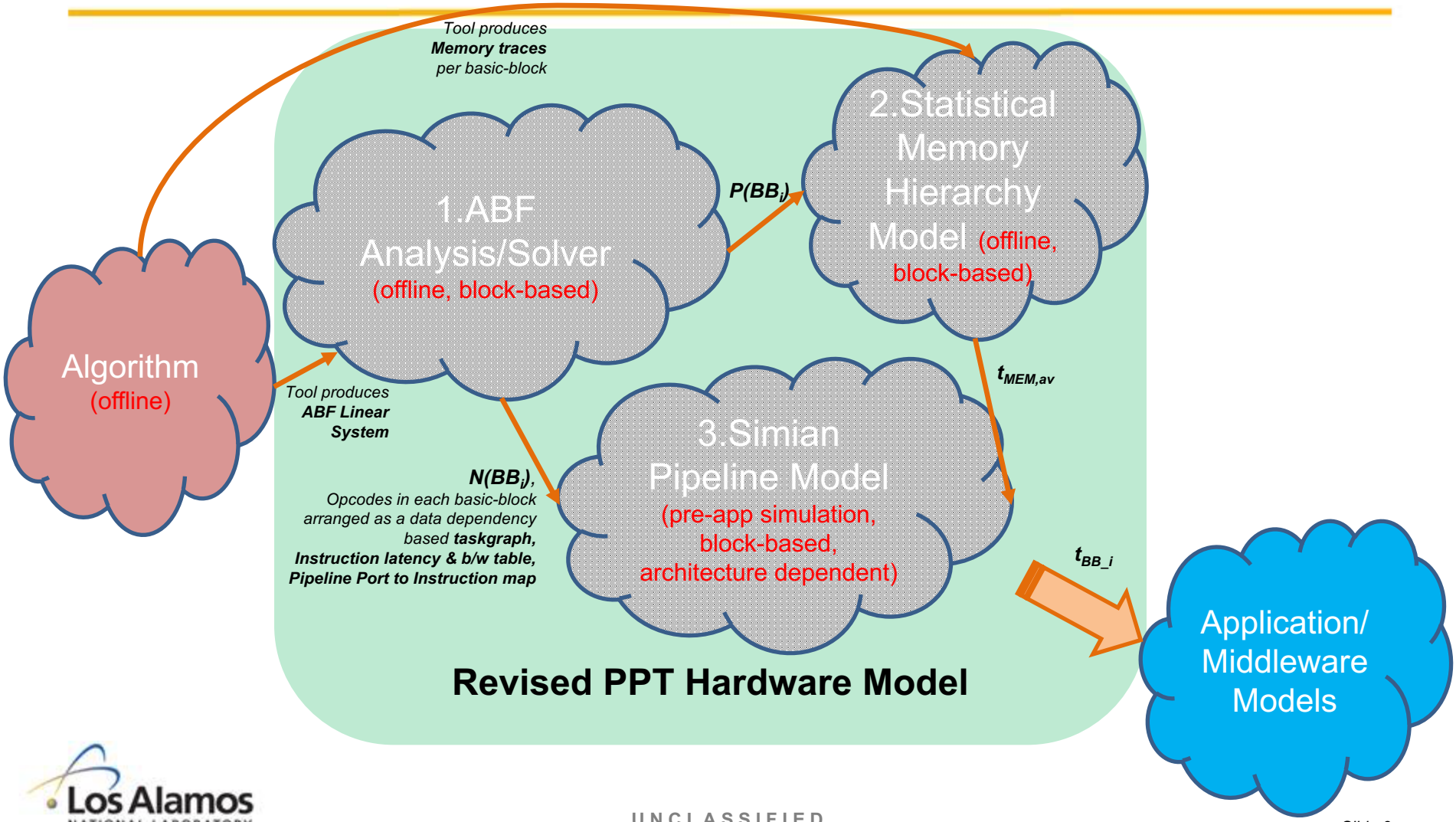
Nandakishore Santhi* (CCS-3)

Gopinath Chennupathi (CCS-3)

Los Alamos National Laboratory (LANL)

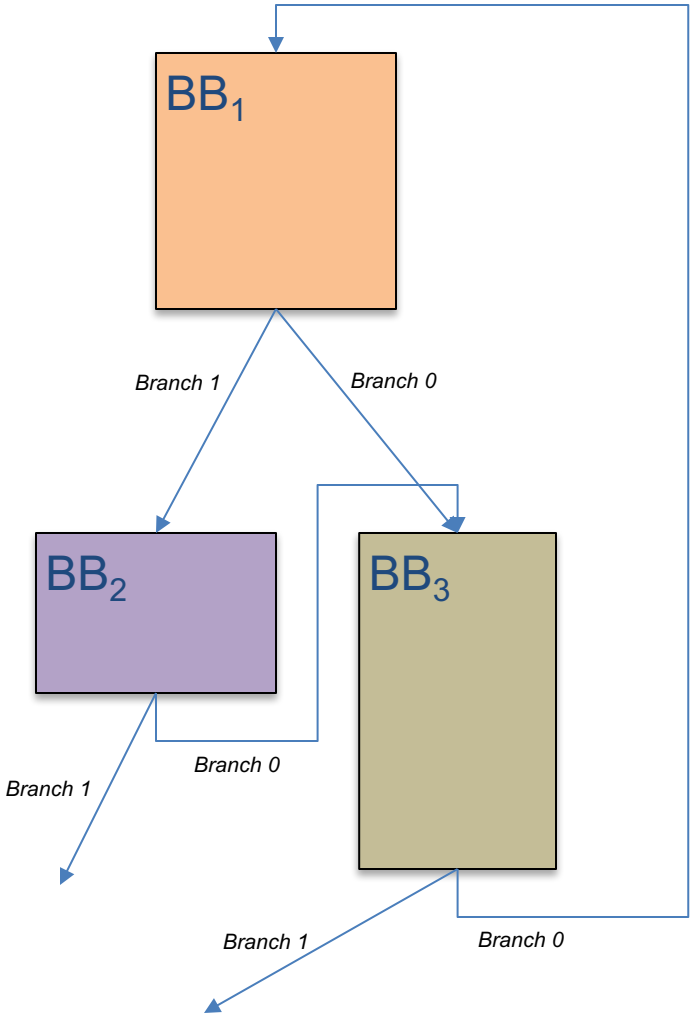
0. Table of Contents

0. Overview: ABF Analysis, Memory, Pipeline



1. Scalable Parametrized Runtime Models

- Static analysis for parametrized models? How can we scalably model:
  - Instruction counts
  - Memory heirarchy effects
  - Hardware pipeline effects
- **Observe**: Relatively easy to count instructions for straight-line programs
- **Generalize**: Basic blocks are straight line (single entry/exit) (practical exeptions are function calls in the middle – those can be specially handled)
  - LLVM can convert high level code to basic-blocks (OR)
  - Manually obtain from psuedo-code
    - Avoid running actual code on physical hardware



1. Average Bracketted Forms (ABF)

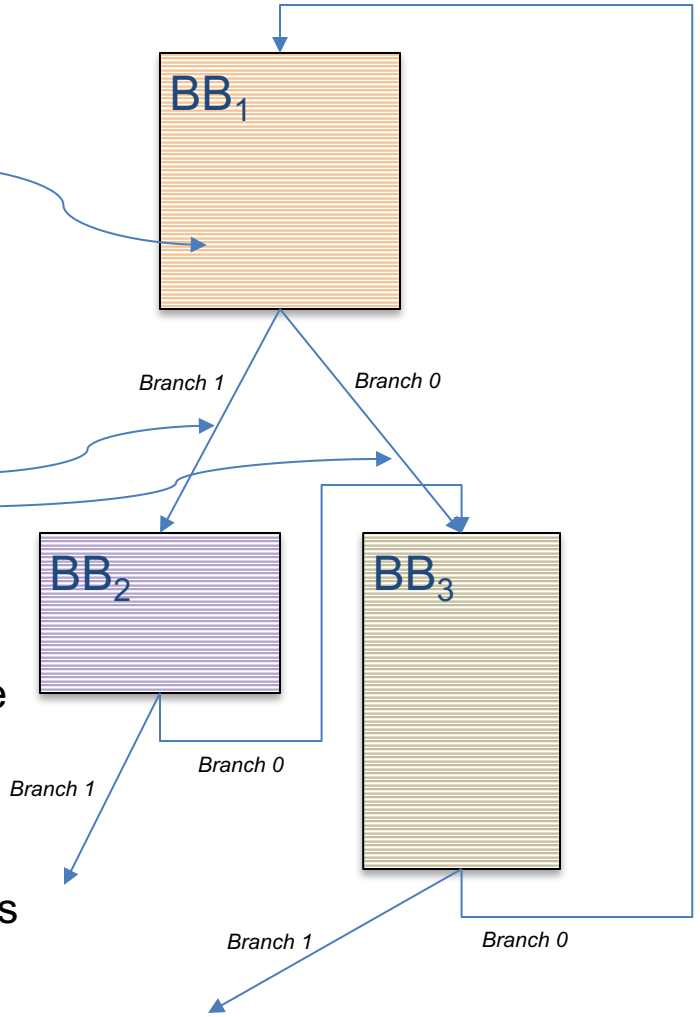
- For straight line programs, a typical bracketted form is (think "*LISP" code*):

  (LOAD, LOAD, ADD, LOAD, MUL, STORE, …)

- For nested straight line programs, one can nest the individual bracketted forms

  (( ), …, (( ), …, ( ), …, ( )), … ( ), …), …

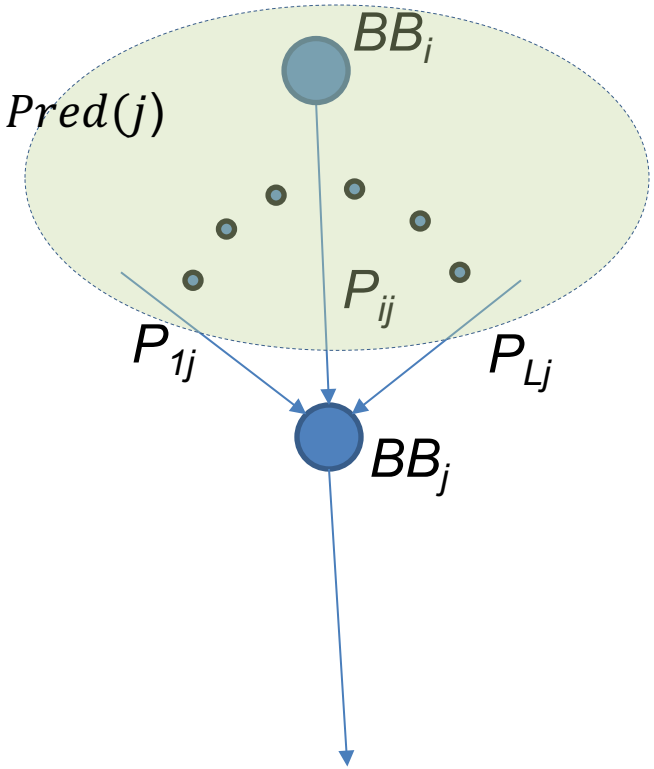
- How do we handle conditional branches?!
- **Our strategy**
  - Assign a runtime probability for each branch in the control-flow graph (were the nodes are straight-line basic-blocks). Prior knowledge and coverage tools can help us assign branching probabilities.
  - Form linear recursive equations connecting the branch probabilities and run-counts for basic-blocks



1. Linear Recursive Equation Model for ABF

- Let a basic block BB_j be executed N_j times during a particular instance of the algorithm's execution
- Let the probability of branching from a predecessor block, BB_i to BB_j be P_{ij}
- Then the basic blocks satisfy the following linear recursive relations:

$$N_j = \sum_{i \in Pred(j)} P_{ij} \, Ni$$



Above is a homogenous square system of linear equations, with many solutions. The main entry block is run once, so add:

$$N_1 = 1$$

1. Example: ABF Equations

Our toolchain automatically produces the following set of ABF equations for a typical benchmark application.

NOTE: Shown here is the per function equation system. We can also do whole-program analysis

....

ABF Analysis Result

Function: *BlkSchlsEqEuroNoDiv*

```
N_1 = 1
N_2 = P_209_9_false * N_1
N_3 = P_209_9_true * N_1
N_4 = N_2 + N_3
```

Function: *bs_thread*

```
N_1 = 1
N_2 = N_1 + N_8
N_3 = P_285_10_false * N_2
N_4 = N_3 + N_6
N_5 = P_290_14_false * N_4
N_6 = N_5
N_7 = P_290_14_true * N_4
N_8 = N_7
N_9 = P_285_10_true * N_2
```

....

ABF Linear Solver Result

Module: blackscholes/src/blackscholes.llvm

Solution:

N_CNDF:1:[label %0] = 3276800 (0.1341353125169)
N_CNDF:2:[label %5] = 1310720 (0.053654125006759)
N_CNDF:3:[label %8] = 1966080 (0.080481187510139)
....

DYNAMIC OP COUNTS FOR THIS INSTANCE:

add : 1687657
alloca:1:i32 : 93388820
and : 2
....
call : 4915201
fadd : 19660800
fcmp : 3276800
fdiv : 6553600
fmul : 58982400
....

Highlighted branching probabilities are set based on the application input scenario, and then the linear system is solved for N_i

1. Average Bracketted Forms (ABF)

- For the overall algorithm, we get an **ABF**:

$$(N_1(\mathbf{BB_1}), N_2(\mathbf{BB_2}), \ldots, N_i(\mathbf{BB_i}), \ldots)$$

- **Observations**
  - Feasible algorithms give unique integer solutions (N_i) to the previous linear system
  - The branching probabilities are input instance dependent
  - Can use this parametrized model to:
    - Count Instructions (opcode) executed at runtime, but without actual runs
    - Model memory hierarchy
    - Model hardware execution pipelines
- **Validation**
  - We implemented automated algorithms to form and solve the linear system from any Clang compiled LLVM IR code as part of the PPT toolkit
  - We have validated our analytic estimates with ByFL instrumented program runs
    - The opcode counts match perfectly

2. Statistical Memory Hierarchy Model

- The average cost of a memory operation depends on:
  - Algorithm metrics: stack/reuse-distance profile
  - Cache metrics: size, latency, bandwidth, associativity at each level
- Average memory access time for a block of size b over a single port interface:
  - Average latency (shown is for 2 level cache, plus RAM):
    $$\boldsymbol{E}(\lambda) = P_{hit1} * \lambda_1 + (1 - P_{hit1})[P_{hit2} * \lambda_2 \; + \; (1 - P_{hit2}) * \lambda_{RAM}]$$
  - Similarly for average throughput rate, $\boldsymbol{E}(\beta)$
  - Average access time, $t_{MEM,av} = (1 - b)\boldsymbol{E}(\beta) + \boldsymbol{E}(\lambda)$
  - Analytical model for direct-cache hit-rate given a stack distance d, and cache parameters such as associativity (A) and block-size (B):
    $$P(hit|d) = \sum_{a=0}^{A-1} \binom{d}{a} \left(\frac{A}{B}\right)^{a} \left(\frac{B - A}{B}\right)^{(d-a)}$$
  - After un-conditioning we get: $P(hit) = \sum_d P(hit|d).P(d)$

2. Stack Distance Profile and ABF

- Stack (reuse) distance, d for a memory address instance is the number of unique other addresses accessed between its previous and current access
- Stack distance profile $P(d)$ is the overall distribution for an algorithm
- $P(d)$ estimation can be time consuming for large memory traces:
  - Naïve implementation is: $O(n^2)$
  - Clever implementations in literature usually: $O(n.log(n))$
- **Our strategy**

$$P(d) = \sum_i P(d|BB_i) \,.\, P(BBi)$$

$$P(BB_i) = \frac{N_i}{\sum_j N_j}$$

Where N_i are eventually obtained by ***solving the ABF linear system***

2. Block Level Sampled Stack Distance Profile

- Conditional reuse profile for each basic-block, $P(d|BB_i)$ can be determined from trial runs and sampling across occurrences of that particular basic-block
- If in addition, $P(d|BB_i)$ remains unchanged with input and branching probabilities, the final reuse profile can be calculated by repeatedly solving for $P(BB_i)$ using different ABF equation systems
- With S samples per basic-block, M static basic-blocks per algorithm, and trace-length n, the worst-case complexity of our stack distance profile algorithm is $O(S.M.n) = O(n)$ for constant S, M
- Preliminary results show good agreement with direct estimation methods

2. Example: Reuse Profile with Sampling

C source code

```
int loop(int A[], int sum, int SIZE)  {
   fprintf(stderr, "SIZE: %d\n", SIZE);
   for(int i=0; i<SIZE; i++) A[i] = 2*i;
   for(int i=SIZE/2; i<SIZE; i++) sum += A[i];
   return sum;
}


int main(int argc, char *argv[]) {
   const int SIZE = 1000;
   int A[SIZE];
   int sum = 1;
   sum = loop(A, sum, SIZE);
}
```

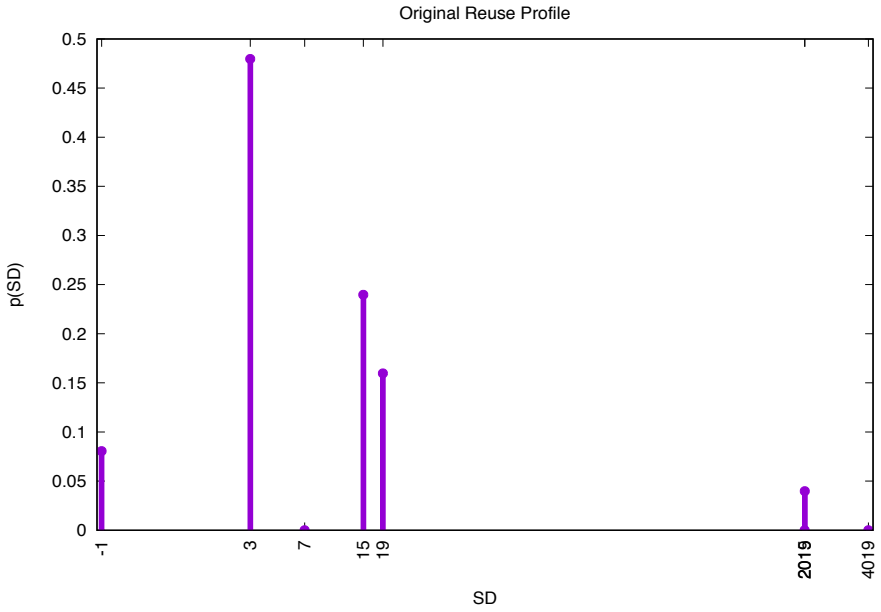
Exec after LLVM instrumentation

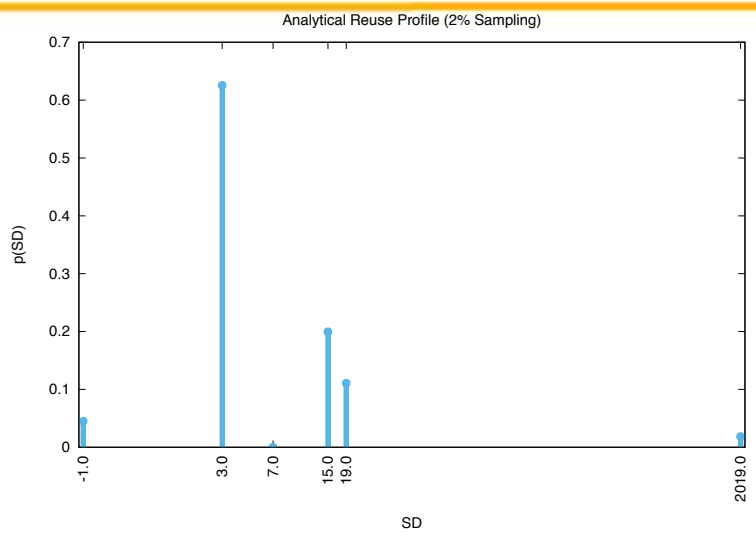
Runtime trace

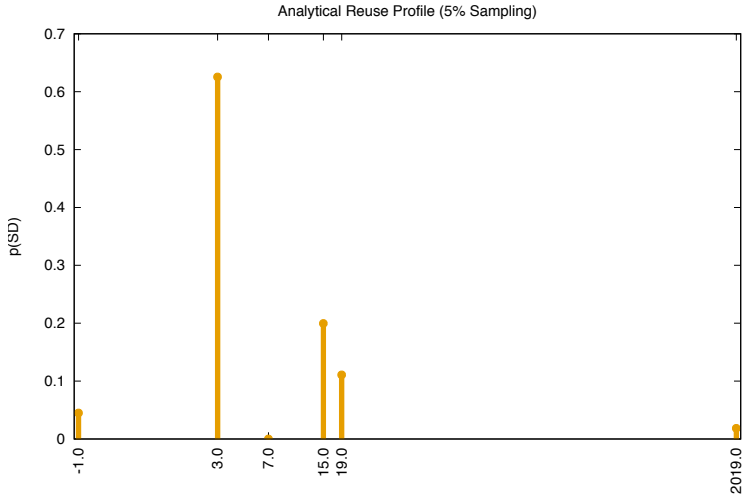
```
...
LD: 0x7ffe9e874e62
ST: 0x7ffe9e874e63
BB DONE: (loop, for.inc)

BB START: (loop, for.cond)
LD: 0x7ffe9e874e60
LD: 0x7ffe9e874e61
LD: 0x7ffe9e874e62
LD: 0x7ffe9e874e63
LD: 0x7ffe9e874e70
LD: 0x7ffe9e874e71
LD: 0x7ffe9e874e72
LD: 0x7ffe9e874e73
BB DONE: (loop, for.cond)

BB START: (loop, for.body)
LD: 0x7ffe9e874e60
ST: 0x7ffe9e874e61
...
```

- We instrumented LLVM/Clang to produce basic-block annotated memory trace at runtime
- The whole trace is sampled for each basic block, and $P(d)$ estimated

2. Example: Reuse Profile with Sampling

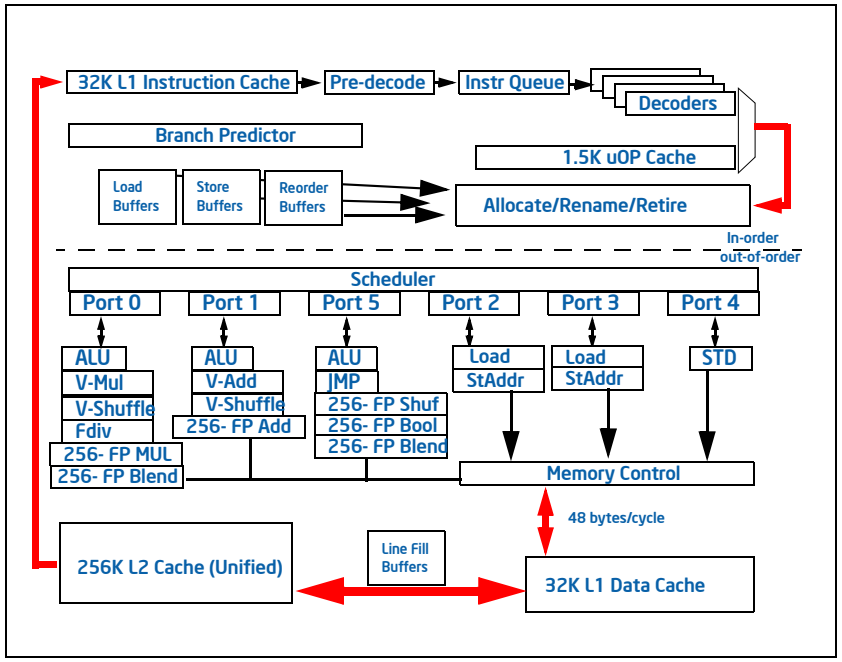






3. Instruction Pipeline Model

- Modern CPUs make use of concurrent hardware pipelines to execute instructions in parallel
- While modeling pipelines, we ignore branch prediction and prefetching



Intel Microarchitecture and Pipeline Functionality: Sandy Bridge Ports and Associated Scheduler (Source: Intel Optimization Reference Manual, 2013)

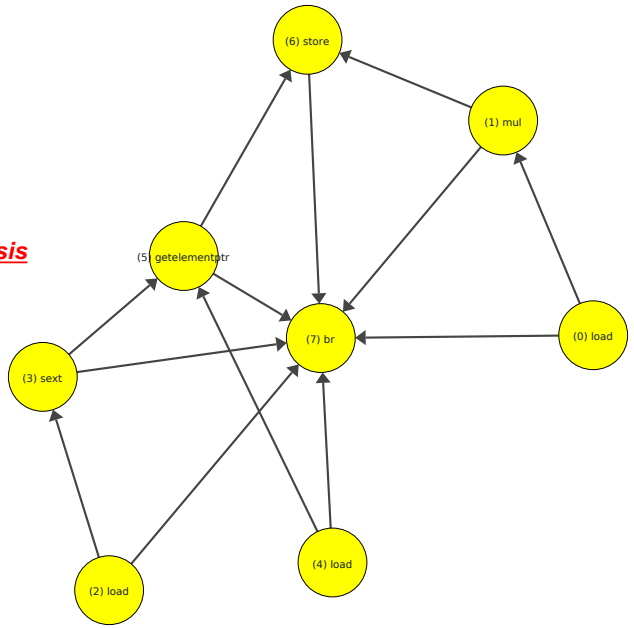
3. A Scalable Pipeline Modeling Strategy

- To retain scalability, we model pipelines at the basic-block level
- Each basic-block is analyzed and a data-flow graph (DAG) is formed
- Going forward, these ***tasklets*** (graphs) will replace current ***tasklists*** in PPT toolkit

```
; <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 inbounds i32, i32* %11, i64 %10
store i32 %8, i32* %12, align 4
br label %13
```

Annotated Basic Block in LLVM IR

Automated Data Dependency Analysis



Tasklet graph for basic-block

3. Block Level Pipeline Modeling Strategy

- A Simian model for the particular CPUs pipeline resources is formed
- The Simian pipeline model takes as input:
  - For each basic-block: the DAG representing its data-dependencies
  - The average memory access time (**LOAD/STORE**) – computed using the statistical memory heirarchy model
  - The latency and throughput for each pipeline port in the CPU architecture
    - This is assumed to remain the same for each instruction supported by that port
- At the start of the application simulation, a Simian pipeline model pre-computes the runtimes per basic-block by running a one-time **sub-simulation** on each basic-block
- Finally, the ABF solution is used to estimate the runtime for entire application (or its parts) as the application simulator runs:

$$(N_1(\mathbf{BB_1}), N_2(\mathbf{BB_2}), \ldots, N_i(\mathbf{BB_i}), \ldots)$$

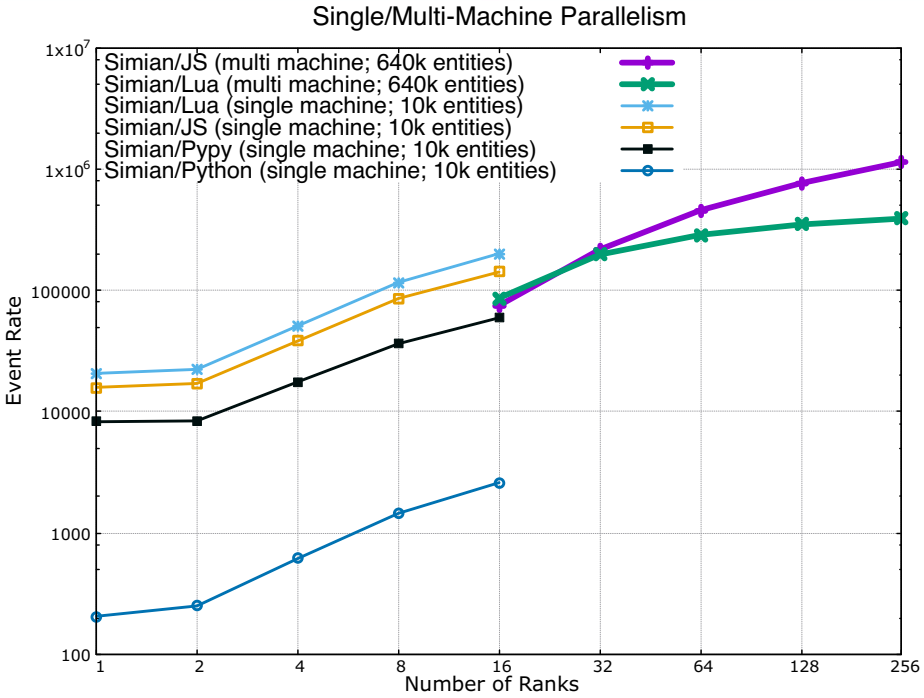
4. Conclusions

- We presented a static analysis technique for parametric modelling of runtime traits of algorithms
- Linear recurrence solution technique was proposed for flexible parametrization of branchy programs (ABF)
- A statistical memory heirarchy model was developed
- A scalable pipeline model was developed
- Both memory heirarchy model and pipeline model uses the ABF solution as input
- We have already implemented most of our algorithms – overall quality of the whole-algorithm runtime prediction remains to be assessed in the coming months, and results will be published

Thank you!

SimianJS under LA-PDES Benchmark Suite

- In barebones tests, SimianJS has reached event rates > 4 million events/s in serial mode on Mac desktops, which is 2X better than SimianLua
- **LA-PDES Benchmark Runs**: SimianLua initially performs better with MPI over multiple machines – due to the efficient, JITed C/FFI in Luajit. At higher entity and rank counts, better memory management and garbage collection pulls SimianJS ahead.



Two scenarios are plotted: (a) Single machine: there are 10,000 entities in total for each rank (b) Multi machine: there are 640,000 entities in total for each rank. A sufficiently large number of entities ensures that strong scaling is apparent at higher rank counts.

SimianJS Architecture, MPI4JS

- SimianJS is implemented over a *LANL customized Mozilla Spidermonkey* open-source codebase
- MPI4JS interface was developed to support JS-native calls to MPI from usual stand-alone (not browser based) Javascript code
- Several other useful C functions are also exposed to the Javascript side
- As a PDES engine, SimianJS user API closely mirrors both SimianLua and SimianPie. Use of MPI for simulation is optional.