Proceedings of the 2017 Winter Simulation Conference
W. K. V. Chan, A. D'Ambrogio, G. Zacharewicz, N. Mustafee, G. Wainer, and E. Page, eds.

AN ANALYTICAL MEMORY HIERARCHY MODEL FOR PERFORMANCE PREDICTION

Gopinath Chennupati
Nandakishore Santhi
Stephan Eidenbenz
Sunil Thulasidasan

Los Alamos National Laboratory
Bikini Atoll Rd., SM 30
Los Alamos, NM 87545, USA

ABSTRACT

As the US Department of Energy (DOE) invests in exascale computing, performance modeling of physics codes on CPUs remain a challenge in computational co-design due to the complex design of processors that include memory hierarchies, instruction pipelining, and speculative execution. We present Analytical Memory Model (AMM), a model of cache memory hierarchy, embedded in the Performance Prediction Toolkit (PPT) – a suite of discrete-event-simulation-based codesign hardware and software models. AMM enables PPT to significantly improve the quality of its runtime predictions of scientific codes.

At its technical core, AMM uses a computationally efficient, stochastic method to predict the distributions of reuse distances of a scientific code, where reuse distance is a hardware architecture-independent measure of the patterns of virtual memory accesses. AMM relies on a stochastic, static basic block-level analysis of reuse distance distributions measured from the memory traces of the scientific applications on small instances. The analytical reuse distribution is useful to estimate the effective latency and throughput of a memory access, which in turn are used to predict the overall runtime of a scientific application.

The experimental results show that the predicted and actual runtimes of two scientific mini-applications (matrix multiplication and Blackscholes) are similar, while AMM is a scalable approach.

1 INTRODUCTION

Software/hardware co-design is a basic tenet of the US Department of Energy's (DOE) exascale computing project (see https://exascaleproject.org/). A comprehensive approach to the computational co-design considers the design spaces of applications and hardware. We view both software and hardware design spaces for a given computational physics application, such as hydrodynamics, as combinatorial, parametrized search spaces, where we apply search techniques to find optimal software/hardware combinations. Such a meta-strategy relies on the existence of parametrized models for hardware and software.

The Performance Prediction Toolkit (PPT) is a library of such hardware and software models that focuses on computational physics codes and real-life high-performance-computing (HPC) architectures. PPT hardware model parameters include clock speed, latency, memory bandwidth, etc. PPT uses these models to predict the runtime of scientific applications through the implementation of stylised pseudo (proxy) code that mimics the loop structure of an application. PPT contains application models for Molecular Dynamics (Zamora et al. 2016), Smoothed Particle Hydrodynamics (Chapuis et al. 2016), radiative transport, Monte Carlo methods, and models for MPI (Ahmed et al. 2016a). On the hardware side, PPT has interconnect models of most standard HPC topologies (Ahmed et al. 2016), and very basic parametrized compute node models for the most-widely used compute nodes such as Intel Haswell.

PPT predictions are reasonably accurate for parallel scientific codes whose performance is determined through MPI or thread-level parallelism. Contrarily, the basic parametrized compute node models are inadequate to predict code performance on a single core, especially, when the advanced CPU features such as memory hierarchy, speculative execution, and instruction level-parallelism play a vital role. For instance, the simple question of how doubling the L_2 cache impact runtime of a given code is unclear in PPT using the simple parametrized hardware models. Interestingly, PPT models for GPU-based architectures (Chapuis et al. 2016) perform better due to the absence of such advanced, non-linear hardware features on GPUs.

We propose an in-depth model of memory hierarchy, which we conjecture to be the most impactful of the advanced CPU features (perhaps leaving pipelining and speculative execution models for future work). Our memory hierarchy model, Analytical Memory Model (AMM), runtime prediction depends on probabilistically measuring the reuse distance distributions from the memory traces of an application at smaller input sizes. Memory traces are generated with basic block labels with LLVM instrumentation. We evaluate AMM on two benchmarks: matrix multiplication (Gunnels et al. 2001) and Blackscholes (Bienia et al. 2008). We predict the reuse distance distributions of larger inputs using the memory trace at smaller input sizes on both the benchmarks. Furthermore, we use these analytical reuse profiles to calculate the effective latency and throughput of a memory access. The analytical latency and throughput help predict the runtimes of the benchmark applications. The results suggest that the characteristic behaviour of predicted runtimes is similar to that of the actual runtimes, scales due to traces at smaller inputs and sampling.

The rest of the paper is organised as follows: section 2.1 presents an overview of PPT, section 2.2 discusses the related work on reuse distance; section 3 presents AMM, section 4 shows the experimental settings and results; section 5 concludes and recommends future research.

2 BACKGROUND

2.1 Performance Prediction Toolkit

The Performance Prediction Toolkit (PPT), is a scalable co-design tool that contains the hardware and middleware models, which accept proxy applications as input in runtime prediction. PPT relies on Simian (Santhi et al. 2015), a parallel discrete event simulation engine in Python or Lua, that uses the process concept, where each computing unit (host, node, core) is a Simian entity. Processes perform their task through message exchanges to remain active, sleep, wake-up, begin and end.

PPT hardware model of a compute core (such as Haswell core) consists of a set of parameters, such as clock speed, memory hierarchies, their sizes, cache-lines, access times for different caches, average cycle counts of ALU operations, etc. These parameters are ideally read off a spec sheet or are learned using regression models devised from hardware counters (PAPI) data. The compute core model offers an API to the software model, a function called $time\_compute()$, takes as input a *tasklist*. A tasklist is an unordered set of ALU, and other CPU-type operations (in particular virtual memory loads and stores). The PPT application model mimics the loop structure of a mini-app and replaces the computational kernels with a call to the hardware model's $time\_compute()$ function.

The key challenge for the hardware model's time_compute-function is to translate virtual memory accesses into actual cache hits and misses. AMM solves this challenge, where our previous alternatives explicitly include the hit-rates as inputs to the tasklists. Explicit hit-rates inevitably only reflect the application modeler's best guess, perhaps informed by a few small test problems using hardware counters; also hard-coded hit-rates make the hardware model insensitive to changes in cache sizes. Alternatively, we use reuse distance distributions in the tasklists. In general, reuse profiles require the application modeler to run a very expensive trace analysis on the real code that realistically can be done at best for small examples.

We propose AMM that predicts the cache hit-rates for a given input using an analytical reuse distance calculation based on stochastic reuse distances of basic-blocks.

2.2 Reuse Distance

The reuse distance of a memory reference (M) is defined as the number of distinct addresses in the trace between the two consecutive accesses to the same reference, M. Reuse profile analysis helps to identify the data locality and the availability of a data item in cache.

The compiler generated trace files for large scale scientific applications are often in tens and/or hundreds of gigabytes (GB). Calculating reuse profiles from such larger files is infeasible, moreover, applications spend enormous computational effort in generating these memory traces. Alternative solutions are: generate the synthetic traces (Agarwal et al. 1989); approximate the reuse distances (Zhong et al. 2003).

In synthetic traces, Partial Markov Model (PMM) (Agarwal et al. 1989) produced sequential and random memory references that rely on the existence of original trace and reported inaccuracies in the reuse profiles. Synthetic traces in (Eeckhout et al. 2000) tried to establish a pattern in the memory references through instruction profiling, branches and dependency analysis. The attempts of Hassan et al. (2007) adapted least recently used stack models (Brehob and Enbody 1999) over PMM states to accurately produce synthetic traces. The reuse profiles of these traces were accurate but not scalable.

In approximating the reuse profiles, Zhong et al. (2003) estimated the reuse patterns of a whole program based on the training runs of a few number of small inputs. In a different attempt, Chatterjee et al. (2001) applied a set of formulas to characterize the cache misses. His model perfectly handled the nested loops and non-linear array layouts. However, Chatterjee's model lack the runtime knowledge of loop bounds. Recent attempt in (Fang et al. 2004) predicted the miss rate per instruction based on reuse distance, however, such a fine grained miss-rate estimation does not scale with the program input.

In contrast to the existing attempts, AMM is simple, scalable and relies on LLVM instrumentation. We calculate reuse profiles for each basic block of a program. These reuse profiles help to measure the hit-rates of a cache at different levels, which further help to estimate the effective latency and throughput, using them predict the runtimes of scientific applications.

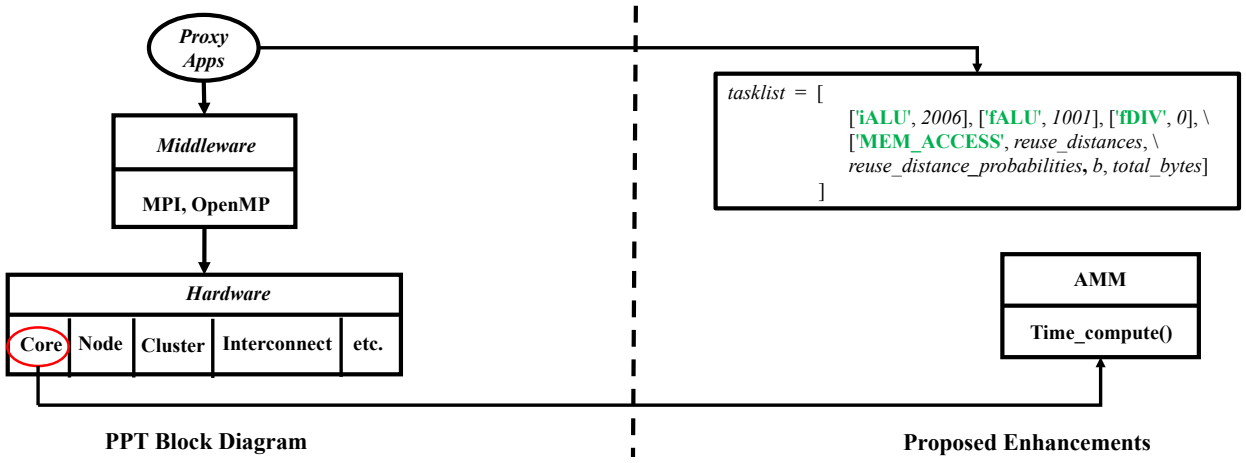

Figure 1: The block diagram of PPT with the enhancements to the hardware models and proxy applications.

3 ANALYTICAL MEMORY MODEL

AMM runtime prediction works in three steps: (a) generate memory trace with basic block labels, (b) calculate the analytical reuse profile of a program from the labeled memory trace, (c) predict the runtime using effective latency and throughput.

Before describing the three steps, we explain how AMM fits in the design of PPT. Figure 1 shows PPT and AMM. AMM replaces the CPU core model in the *hardware* layer of PPT, as a result, the functionality of estimating the time varies from that of the existing time compute function. Another difference is in

the proxy applications, the items in the *tasklist* vary from the earlier models. The proposed enhancements to the tasklist items include the reuse distance distribution (reuse distances and their probabilities), block size (b, is a contiguous block of memory, explained later in section 3.3) and the total memory required for the program. The reuse distributions replace the explicit hit-rates of the tasklist, thereby the hit-rates are measured using the reuse profile. These parameters change with respect to the input size of the program. However, AMM calculates the reuse profile of a program from the labeled memory trace at smaller input size, the same memory trace is used to calculate the reuse profiles at larger inputs.

```
#include <stdio.h>
#define SIZE 1000

int main () {
static int A[SIZE];
int sum = 0, i;
    for(i=0; i<SIZE; i++) {
        A[i] = 2*i;
        sum += A[i];
    }
}
```

(a) Example program

```
BB: (main, entry)

BB: (main, body)

BB: (main, for_cond)

BB: (main, for_body)

BB: (main, for_inc)

BB: (main, for_end)
```

(b) Basic blocks (BB)

```
BB Start:  (main, entry)
LOAD: 0x7ffe5d1d8db0
BB End:  (main, entry)
BB Start:  (main, for_cond)
LOAD: 0x7ffe5d1d8dd0
STORE: 0x604100
BB End:  (main, for_cond)
BB Start:  (main, for_body)
LOAD: 0x7ffe5d1d8db3
STORE: 0x7ffe5d1d8db
BB End:  (main, for_body)
```

(c) BB memory trace

Figure 2: (a) An example C program that calculates the sum of elements of an array, (b) and (c) represent the corresponding basic block (BB) labels and the BB labeled memory trace respectively.

3.1 Generate Memory Trace

AMM generated memory trace of a program contains of named basic block labels to identify the memory addresses that belongs to the corresponding BB. The basic block labels in the memory trace of a program are generated using an LLVM application characterization tool, Byfl (Pakin and McCormick 2013), developed at LANL. We tweaked Byfl to get the BB names and the memory addresses. Figure 2(a) shows an example program in C, Figure 2(b) represents the BB names of the program. Figure 2(c) represents the BB labeled memory trace resulted from Byfl. In the labeled memory trace, a basic block (BB_i) contains all the memory addresses that are generated from executing the code of BB_i. Similar traces can be generated with *valgrind*, however, Byfl is a better choice as it offers hardware-independent application characterization.

3.2 Analytical Reuse Profile

In order to calculate the reuse profiles, we employ a stack distance based cache model (SDCM) (Brehob and Enbody 1999), which identifies the probability of getting a hit at a given cache (for example, L_1, L_2, and L_3) for a given memory reference with a specific reuse distance. The following formula represents the conditional probability of a hit at a given reuse distance ($P(h|D)$):

$$P(h|D) = \sum_{a=0}^{A-1} \binom{D}{a} \left(\frac{A}{B}\right)^a \left(\frac{B-A}{B}\right)^{(D-a)}$$

where D is reuse distance, A is the associativity and B is cache size in terms of number of blocks (which is cache size over cache line size. For example, an L_1 cache of size $64K$ with line size 64 has 1024 blocks). For a direct-mapped cache, $P(h|D)$ is $((B-1)/B)^D$. The unconditional probability of a hit $P(h)$ is:

$$P(h) = \sum_{i=0}^{N} P(D_i) \times P(h|D_i) \tag{1}$$

where, $P(D_i)$ is the probability of i^{th} reuse distance (D) in a reuse distribution $P(D)$.

3.2.1 Basic block analysis

We calculate the conditional reuse profiles at a given basic block, $P(D|BB_i)$. The input is the i^{th} basic block name and the labeled memory trace. In general, BB_i can appear more than once in the labeled memory trace. From these occurrences of BB_i, we randomly select *sample_size* (user-defined) number of samples, termed *windows*, where each window contains the start and end indexes of a basic block occurrence. Using these indexes, we refer to the labeled memory trace to get the corresponding BB addresses. For each address in the window, we measure the reuse distance (Algorithm 1), that way, each address in all the sampled windows will have one reuse distance value. Finally, we calculate the unique reuse distances and their corresponding probabilities, which is the conditional reuse profile of BB_i, $P(D|BB_i)$.

Algorithm 1 Calculate the reuse distances

```
1:  procedure get_reuse_dist(window, memory_trace)
2:      reuse_dist ← [ ]
3:      for idx, addr in enumerate(window) do
4:          window_trace ← memory_trace[: idx]; dict_rd ← { }; addr_found ← False
5:          for w_adx in range(len(window_trace)) do
6:              w_addr ← window_trace[−w_adx − 1]                ▷ max back reference
7:              if addr == w_addr then addr_found ← True; break
8:              end if
9:              dict_rd[w_addr] = True
10:         end for
11:         if addr_found then reuse_dist.append(len(dict_rd))
12:         else reuse_dist.append(−1)
13:         end if
14:     end for
15:     return reuse_dist
16: end procedure
```

Algorithm 1 calculates the reuse distance of a window. For each address in the window, we refer back into the original trace from the current address till we encounter the same address, termed as *max back reference*, hitherto, the number of unique memory addresses is the reuse distance of the current address in the window. At the end, the algorithm returns a list of all the reuse distances of the window.

3.2.2 Reuse profile of a program

With the conditional reuse profiles of a basic block ($P(D|BB_i)$), we estimate the the final unconditional reuse profile of a program as shown in Eq. 2

$$P(D) = \sum_{i=0}^{n(BB)} P(BB_i) \times P(D|BB_i) \tag{2}$$

where, $n(BB)$ is the number of basic blocks in a program, $P(BB_i)$ represents the probability of executing the i^{th} basic block. We measure $P(BB_i)$ through an off-line static analysis on a given program. Note that the basic blocks with no memory access in their trace have negligible contribution to the reuse profile.

3.2.3 Apriori Probability of a basic block

Let us consider, BB_1, BB_2, …, BB_j, …, BB_{n-1}, BB_n is a series of basic blocks, each basic block has a single entry and exit. Any BB can execute any other BB, for example, if the basic blocks BB_1, BB_2, …, BB_k execute BB_j then the basic blocks BB_1, …, BB_k are treated as the predecessors of BB_j. Therefore,

the basic blocks satisfy the linear recursive relation: $N_j = \sum_{i \in Pred(j)} P_{ij} \times N_i$ where, P_{ij} is the probability of branching (measured off-line using LLVM/GCC coverage analysis) from predecessor block BB_i to BB_j, N_i and N_j are the number of calls to the i^{th} and j^{th} basic blocks. N_j is a homogeneous system of linear equations with many solutions. For most of the source codes, the entry basic block is executed once, hence, $N_1 = 1$. Therefore, the apriori probability of a basic block is represented as shown in Eq. 3,

$$P(BB_j) = \frac{N_j}{\sum_{i=0}^{n(BB)} N_i} \tag{3}$$

where, $n(BB)$ is the total number of basic blocks of an input program. From Eq. 2 and Eq. 3, we get the unconditional reuse profiles for all the basic blocks of a program. Finally, combining these unconditional reuse profiles results in the final reuse distribution of a program.

Note, the probabilities of the individual basic blocks change with the input size. Nevertheless, we use the same memory trace at the smaller input size to estimate the reuse profiles of the large inputs of the program. The off-line analysis with LLVM coverage tool generates the probabilities of bigger inputs.

3.3 Predict runtime

The total predicted runtime of a program is the sum of the average memory access time of a program (T_{avg_mem}) and the time taken for the CPU operations (T_{CPU_ops}) The total memory required for the program and the number of CPU operations are counted using Byfl. The predicted runtime is, $T_{pred} = T_{avg\_mem} + T_{CPU\_ops}$. The average memory access time of a program is shown in Eq. 4:

$$T_{avg\_mem} = \frac{\lambda_{avg} + (b-1) \times \beta_{avg}}{b} \times total\_mem \tag{4}$$

where λ_{avg} is the effective latency, β_{avg} is the effective throughput and b is block size. The latency and throughput are per one memory access, whereas for block size we consider word size with the assumption of the availability of contiguous memory. $total\_mem$ is the total memory required by the program.

The measured analytical reuse profiles are used to estimate the cache hit-rates at different hierarchies (Eq. 1), with which, we measure the effective latency and throughput of a given program. The average latency for a two-level cache is shown in Eq. 5

$$\lambda_{avg} = P_{L_1}(h) \times \lambda_{L_1} + (1 - P_{L_1}(h)) \left[P_{L_2}(h) \times \lambda_{L_2} + (1 - P_{L_2}(h)) \times \lambda_{RAM} \right] \tag{5}$$

where, λ_{L_1}, λ_{L_2}, and λ_{RAM} are the latencies of L_1, L_2 caches and RAM; $P_{L_1}(h)$ and $P_{L_2}(h)$ are the probability of hits for L_1 and L_2 cache respectively. Similarly, we measure the average throughput, β_{avg}. Note the throughput herein is the reciprocal bandwidth. AMM is similar to Aspen (Spafford and Vetter 2012), in a broader sense that both the systems account for the system and software characteristics. However, the two tools are different in their functionality, where Aspen predicts the performance through domain specific language modeling while that of AMM depends on LLVM basic blocks.

4 EXPERIMENTS

In this section, we evaluate the proposed method, AMM. We conduct the experiments on two benchmark applications: matrix-matrix multiplication (Gunnels et al. 2001) that multiplies two matrices; and Blackscholes (Bienia et al. 2008) that predicts the European stock price options through partial differential equations. The source code of the matrix multiplication follows *ijk* approach (has 3 nested loops), whereas that of Blackscholes contains two nested loops and tries to solve a differential equation. The input to the matrix multiplication is initialized with-in the source code, while Blackscholes reads the input from a file.

The AMM hardware and software parameters used in the experiments are described as follows. The hardware parameters are: cache latencies, cache sizes, cache line sizes, associativity, and reciprocal throughput at different cache levels (we use three cache levels both in simulation and real time), RAM

latency, and data bus width. Note that the hardware parameters are measured for a given hardware, for example, Intel and other reliable sources[1] provide these measured values. We can measure these parameters using standard benchmarks, however, that is not our objective in this paper. The software parameters are: total memory of an application, number of integer and floating point operations (addition, multiplication, etc.), block size (see Eq. 4 in section 3.3), measured using Byfl.

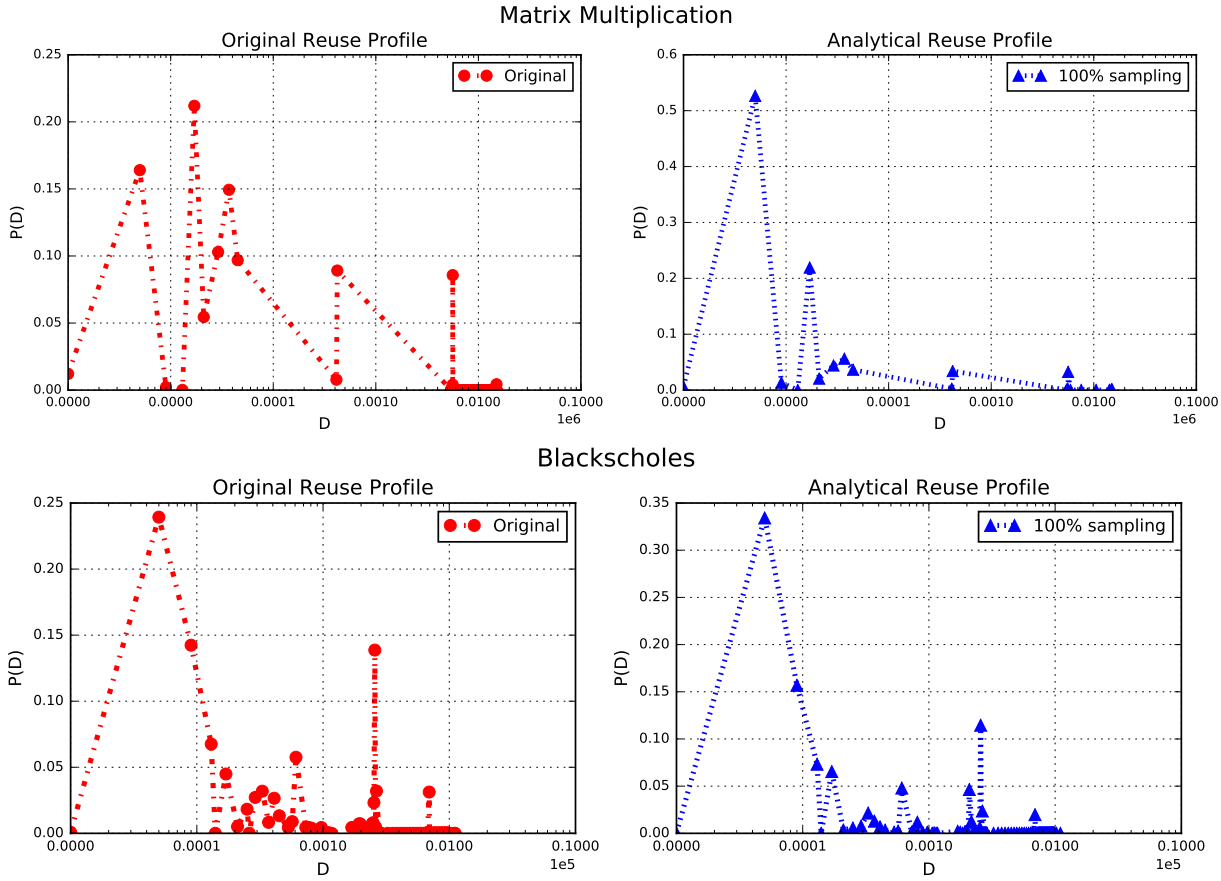

Figure 3: Comparison of the original (left) and analytical (right) reuse profiles of matrix multiplication (top) at an input size of 25 elements and Blackscholes (bottom) at an input size of 16 data points. Original distribution is measured from the original memory trace, while that of the analytical is measured using AMM with 100% sampling. The reuse distance (D) is on *log* scale while P(D) is on decimal scale.

The validation of AMM is two folded: 1) validate the reuse profiles – compare the real and predicted reuse profiles, 2) validate the runtimes – compare the real and predicted runtimes. Both the predicted and actual runs are computed on a single core of a CPU.

4.1 Validate Reuse Profile

Our goal is to validate the analytical reuse profiles against the actual reuse profiles. Although the reuse profile distributions are discrete, in this paper, we present them as continuous for better legibility.

Figure 3 compares the actual and the analytical reuse profiles of both the benchmarks. The analytical reuse profiles are prepared with 100% sampling. For example, if a basic block contains ten occurrences, all of them contribute to calculate the conditional reuse profiles before multiplying with the apriori probability

[1] http://www.7-cpu.com/cpu/Haswell.html

($P(BB_i) \times P(D|BB_i)$) of the corresponding basic block. On both the benchmarks, the reuse distances (D, on x-axis) of the actual and the analytical reuse profiles are identical, so does their frequency. The respective probabilities ($P(D)$, on y-axis) are approximately similar, the analytical probabilities are slightly higher at times because of their dependence on the accuracy of $P(BB_i)$, which is measured through an offline analysis. Nonetheless, such inaccuracies have insignificant impact on the final cache hit-rate, therefore, the analytical reuse distribution is approximately close to that of the actual distribution.

The original reuse profiles are measured using a naive stack (Mattson et al. 1970) based implementation that has a time complexity of $O(NM)$. The analytical reuse profiles are measured as shown in section 3.2, that has a computational complexity of $O(NSB) \sim O(N)$, since the number of samples (S) and size of the basic block (B) are fixed. The worst case complexity is $O(NM)$, iff the *sampling rate* is 100%, which should never happen, since sampling rate of 1% is sufficient to accurately approximate the reuse profiles.

4.2 Validate Runtime

We validate the actual runtimes with that of the predicted for both the benchmarks at different input sizes. The input sizes for matrix multiplication vary from 25, 50, 100, and 200, which are the number of elements in each matrix. The input for Blackscholes is a dataset, each data-point having eight numerical values and a character (price option type). The dataset size varies from 16, 32, 64, and 128 data-points respectively.

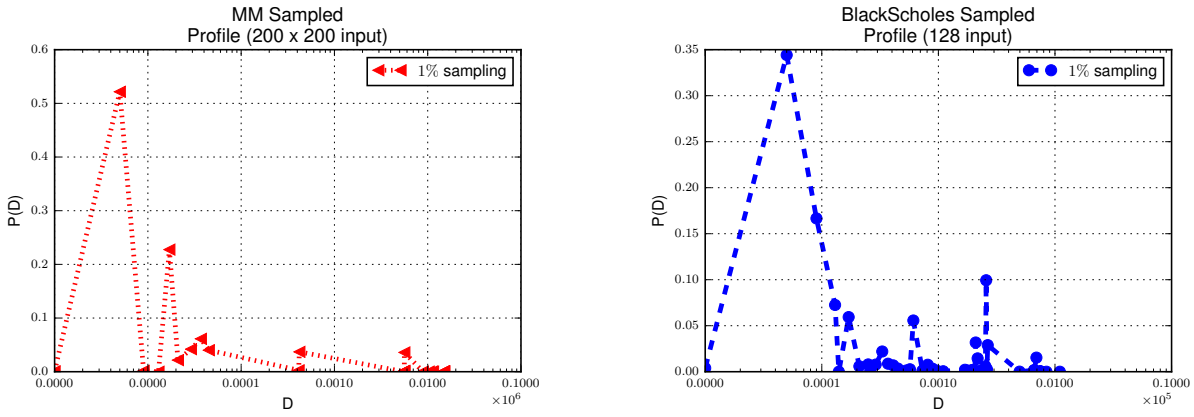

Figure 4: Randomly sampled analytical reuse profiles of both the benchmarks: matrix multiplication (left) and Blackscholes (right) at input sizes of 200 x 200 matrix elements and 128 data points respectively. The rate of sampling is 1%, while the basic block labeled memory traces for both the problems are considered when the *input sizes* are 25 and 16 respectively. The reuse distance (D) is on a *log scale*.

Both actual and predicted runtimes are measured on a Haswell (Intel Xeon E5-2650) core that has a three level cache. In predicting the runtimes, we measure the analytical reuse profiles of all the inputs, considering the memory trace for 25 as the base trace for matrix multiplication, and 16 as the base trace for all the input sizes of Blackscholes. The probabilities of the basic blocks differ as the input size changes, which are measured through an offline static analysis (described in section 3.2.3).

Figure 4 shows the analytical (*sampling rate* of 1%) reuse profiles of both the benchmarks at different input sizes, using the smaller traces as the reference. The randomly sampled reuse profiles are approximately similar to that of the original. However, some large reuse distances disappear due to random sampling, that is, the memory addresses that stay far apart are omitted. For example, if some basic block appears at the bottom of the memory trace is omitted in the random sampling, and that contains memory addresses that appear somewhere at the start of the trace then those larger stack distances will be ignored. Although some large reuse distances are ignored that will have negligible effect on the overall reuse profile, iff the probability of such large reuse distances is relatively small compared to the rest of the reuse distances in

the profile. On the other hand, if they occur more often, there is a high chance that random sampling will select a few of those occurrences as they are part of the repeated basic blocks.

Interestingly, the reuse distance metric is based on virtual memory accesses. However, most modern processors are physically addressed where, memory blocks that are far apart in the virtual address space can be contiguously placed in physical memory, which might perturb the reuse distance. One solution is to model the memory mapping that accurately estimates the distance, one of our future directions.

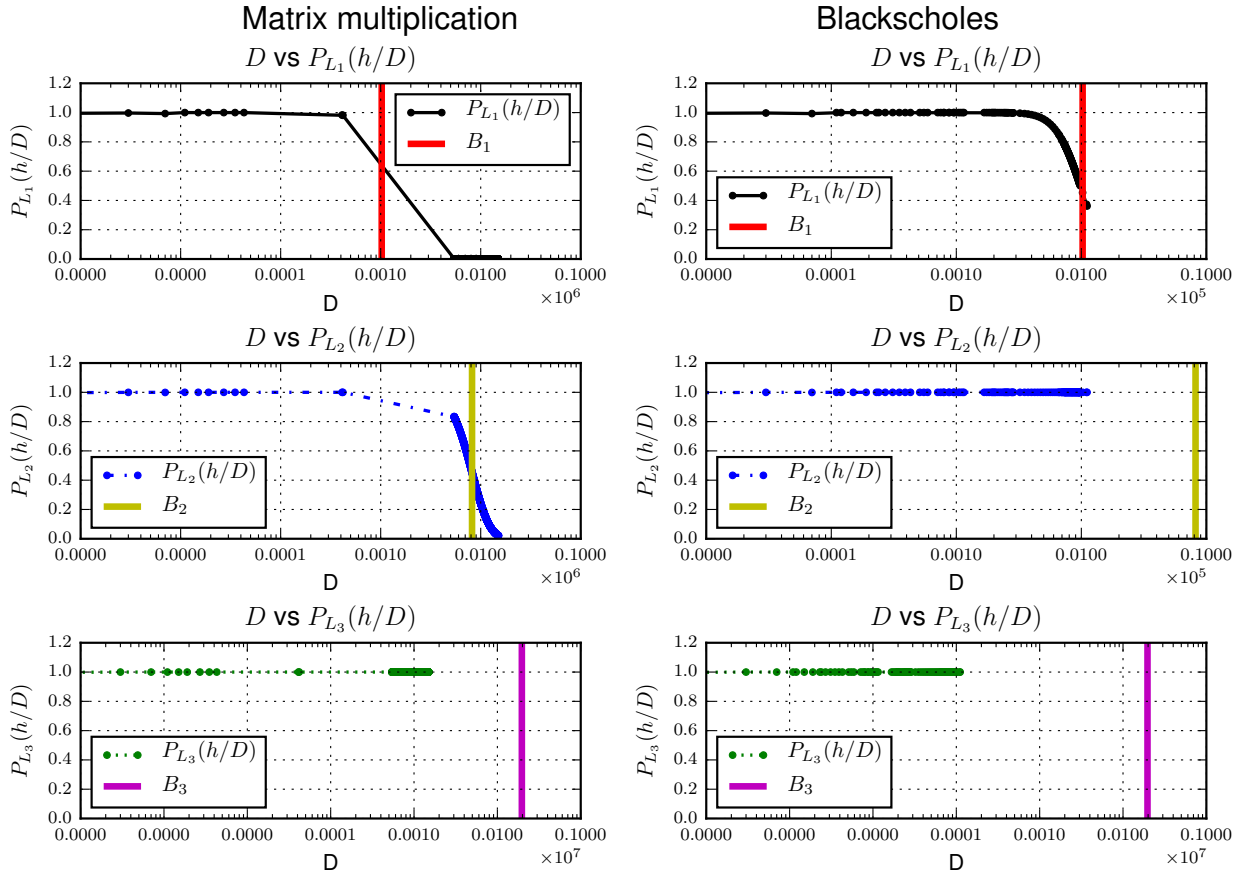

Figure 5: Conditional reuse profiles of both the benchmarks – matrix multiplication (left) and Blackscholes (right). B_1, B_2 and B_3 are the number of blocks (see section 3.2) for L_1, L_2 and L_3 caches respectively.

Figure 5 shows the conditional cache hit-rates at a given reuse distance for three different cache sizes (L_1, L_2, and L_3). The results are for an input size of 25 and 16 on matrix multiplication and Blackscholes respectively. The reuse distance calculation is independent of the underlying hardware architecture. Architecture independent in the sense that the hit-rates adapt to the changes in the cache properties such as the associativity, line size and cache size, etc (see $P(h|D)$ in Eq. 1). Therefore, cache hit-rates at different cache sizes are measured using the same reuse profile. The reuse distance (D) is on a *log scale*, whereas B_1, B_2 and B_3 are the number of blocks (cache size/cache line size). On both the benchmarks, the cache hit-rate at a stack distance ($P_{L_1}(h/D)$) suddenly drops, for example, on L_1 cache observe the drop after the number of blocks (B_1). A similar behaviour is found for L_2 cache in the case of matrix multiplication but not on Blackscholes. On the other hand, from Figure 4, we observe that the probabilities of reuse distance ($P(D_i)$) at large reuse distances are approximately zero, which hints that the contribution of those reuse distances is negligible. With these two unique characteristics, reuse distances that exceed the cache sizes are always a miss. From the results in Figure 5, the entire data fits in L_3 for

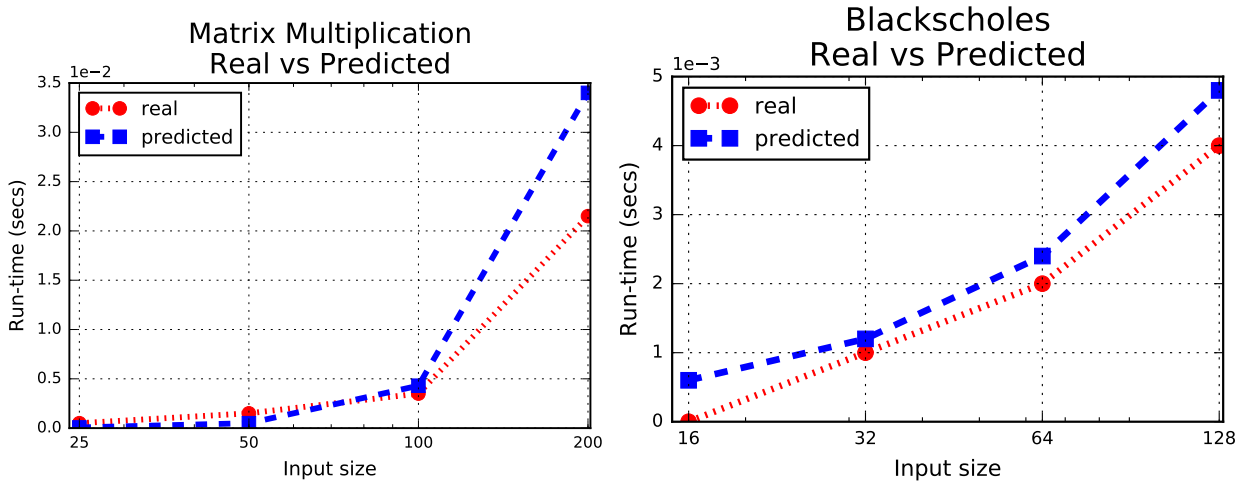

Figure 6: Real versus predicted runtimes (in seconds) of matrix multiplication (left) and Blackscholes (right) at various input sizes.

both the benchmarks, a discussion on the locality of data is out of our scope. However, the 1% sampled reuse profiles accurately approximate the availability of the data on a given cache hierarchy, which can be used in estimating the effective latency and throughput, thereby the runtime of an application.

We validate the predicted runtimes (using the reuse profiles in Figure 4) of both the benchmarks at different input sizes. Figure 6 compares the actual runtimes with that of the predicted on both the benchmarks. On both the benchmarks, the non-trivial observation is that the runtimes are over-predicted with respect to the actual runtimes, especially, at larger input sizes. Although we over-predict, the characteristic behaviour of the runtimes when compared with the input remains in coherence with the actual runtime. The reason behind over-prediction lies in the fact that AMM is purely a memory model. AMM assumes the execution of the program in complete sequential mode, whereas the actual CPU core executes the independent tasks simultaneously through pipeline. The pipeline handles more work-load within the same CPU cycle. Along with pipelines factors such as prefetching, replacement strategy, hardware threads, speculative execution models, etc., contribute to reduce the computational effort of an application. Building a parametrised model (one of our future directions) with these factors that work together with AMM would reduce the over-prediction in runtimes.

5 CONCLUSION

We presented a novel analytical memory model (AMM) that produces basic block labeled memory traces using LLVM tools. The memory traces at smaller inputs are randomly sampled to produce the reuse distance distributions of scientific applications. Using the smaller input memory traces, reuse profiles of the applications are estimated at various larger input sizes. The analytically measured reuse profiles are similar to that of the actual reuse profiles. With these reuse profiles, we showed the availability of data (cache hit-rates) for the processor through various cache hierarchies. Furthermore, the cache hit-rates at different cache hierarchies are useful for measuring the effective latency and throughput per one memory access of an application. Finally, the CPU operations, effective latency and throughput are used to predict the runtimes of the applications. The characteristic behaviour of the predicted runtimes is similar to that of the actual runtimes. However, one drawback of AMM is that the predicted runtimes are slightly higher than that of the actual, such an over-prediction is due to the fact that AMM does not have parametrised models for pipelines, and speculative execution models. Developing and integrating these missing models would guarantee a close prediction, therefore, is one of our future directions. Another interesting research

direction is the use of AMM in combination with the distributed models, similar to (Ahmed et al. 2016b, Gianni et al. 2008) for the performance prediction of large scale scientific applications.

REFERENCES

Agarwal, A., J. Hennessy, and M. Horowitz. 1989. "An Analytical Cache Model". *ACM Transactions on Computer Systems* 7 (2): 184–215.

Ahmed, K., J. Liu, S. Eidenbenz, and J. Zerr. 2016. "Scalable Interconnection Network Models for Rapid Performance Prediction of HPC Applications". In *18th IEEE International Conference on High Performance Computing and Communications; 14th IEEE International Conference on Smart City; 2nd IEEE International Conference on Data Science and Systems, HPCC/SmartCity/DSS 2016*, 1069–1078. Sydney, Australia.

Ahmed, K., M. Obaida, J. Liu, S. Eidenbenz, N. Santhi, and G. Chapuis. 2016a. "An Integrated Interconnection Network Model for Large-Scale Performance Prediction". In *Proceedings of the 2016 annual ACM Conference on SIGSIM Principles of Advanced Discrete Simulation, SIGSIM-PADS 2016*, 177–187. Banff, Alberta, Canada.

Ahmed, K., M. Obaida, J. Liu, S. Eidenbenz, N. Santhi, and G. Chapuis. 2016b. "An Integrated Interconnection Network Model for Large-Scale Performance Prediction". In *Proceedings of the 2016 Annual ACM Conference on SIGSIM Principles of Advanced Discrete Simulation*, SIGSIM-PADS '16, 177–187. New York, NY, USA: ACM.

Bienia, C., S. Kumar, J. P. Singh, and K. Li. 2008. "The PARSEC Benchmark Suite: Characterization and Architectural Implications". In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, 72–81. New York, NY, USA: ACM.

Brehob, M., and R. Enbody. 1999. "An Analytical Model of Locality and Caching". *Tech. Rep. MSU-CSE-99-31*.

Chapuis, G., S. Eidenbenz, and N. Santhi. 2016. "GPU Performance Prediction Through Parallel Discrete Event Simulation and Common Sense". *EAI Endorsed Trans. Ubiquitous Environments* 3 (10): e4.

Chapuis, G., D. Nicholaeff, S. Eidenbenz, and R. S. Pavel. 2016. "Predicting Performance of Smoothed Particle Hydrodynamics Codes at Large Scales". In *Proceedings of the 2016 Winter Simulation Conference, WSC*, edited by T. M. K. Roeder, P. I. Frazier, R. Szechtman, E. Zhou, T. Huschka, and S. E. Chick, 1825–1835. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Chatterjee, S., E. Parker, P. J. Hanlon, and A. R. Lebeck. 2001. "Exact Analysis of the Cache Behavior of Nested Loops". In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, 286–297. New York, NY, USA: ACM.

Eeckhout, L., K. de Bosschere, and H. Neefs. 2000. "Performance Analysis Through Synthetic Trace Generation". In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '00, 1–6. Washington, DC, USA: IEEE.

Fang, C., S. Carr, S. Önder, and Z. Wang. 2004. "Reuse-distance-based Miss-rate Prediction on a Per Instruction Basis". In *Proceedings of the 2004 Workshop on Memory System Performance*, MSP '04, 60–68. New York, NY, USA: ACM.

Gianni, D., A. D'Ambrogio, and G. Iazeolla. 2008. "A Layered Architecture for the Model-driven Development of Distributed Simulators". In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, Simutools '08, 61:1–61:9. ICST, Brussels, Belgium, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

Gunnels, J. A., G. M. Henry, and R. A. V. D. Geijn. 2001. "A Family of High-Performance Matrix Multiplication Algorithms". In *Proceedings of the International Conference on Computational Sciences-Part I*, ICCS '01, 51–60: Springer.

Hassan, R., A. Harris, N. Topham, and A. Efthymiou. 2007. "Synthetic Trace-Driven Simulation of Cache Memory". In *21st International Conference on Advanced Information Networking and Applications Workshops*, Volume 1 of *AINAW '07*, 764–771.

Mattson, R. L., J. Gecsei, D. R. Slutz, and I. L. Traiger. 1970. "Evaluation Techniques for Storage Hierarchies". *IBM Systems Journal* 9 (2): 78–117.

Pakin, S., and P. McCormick. 2013. "Hardware-Independent Application Characterization". In *International Symposium on Workload Characterization (IISWC)*, 111–112. Portland, Oregon, USA: IEEE.

Santhi, N., S. Eidenbenz, and J. Liu. 2015. "The Simian concept: Parallel Discrete Event Simulation with interpreted languages and just-in-time compilation". In *Proceedings of the 2015 Winter Simulation Conference (WSC)*, edited by L. Yilmaz, W. K. V. Chan, I. Moon, T. M. K. Roeder, C. Macal, and M. D. Rossetti, 3013–3024. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Spafford, K. L., and J. S. Vetter. 2012. "Aspen: A Domain Specific Language for Performance Modeling". In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, 84:1–84:11. Los Alamitos, CA, USA: IEEE.

Zamora, R. J., A. F. Voter, D. Perez, N. Santhi, S. M. Mniszewski, S. Thulasidasan, and S. J. Eidenbenz. 2016. "Discrete Event Performance Prediction of Speculatively Parallel Temperature-Accelerated Dynamics". *Simulation* 92 (12): 1065–1086.

Zhong, Y., S. G. Dropsho, and C. Ding. 2003. "Miss Rate Prediction Across All Program Inputs". In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, PACT '03, 79–91. Washington, DC, USA: IEEE.

AUTHOR BIOGRAPHIES

GOPINATH CHENNUPATI is a Post-doctoral Research Associate with the Information Sciences Group (CCS3) at Los Alamos National Laboratory, New Mexico, United States of America. He holds a Ph.D. in computer science from University of Limerick, Ireland. His research interests include parallel discrete event simulation, performance modeling, high performance computing, machine learning, and artificial intelligence. His email address is gchennupati@lanl.gov.

NANDAKISHORE SANTHI is a Computer Research Scientist with the Information Sciences Group (CCS-3) at Los Alamos National Laboratory. He holds a PhD in Electrical and Computer Engineering from the University of California San Diego. His areas of research interests include parallel discrete event simulation, performance modeling of HPC systems, applied mathematics, communication systems and computer architectures. His email address is nsanthi@lanl.gov.

STEPHAN EIDENBENZ is the Director of the Information Science and Technology (ISTI) institute at Los Alamos National Laboratory. He obtained a PhD from the Swiss Federal Institute of Technology, Zurich (ETHZ) in Computer Science. His research interests include cyber security, computational co-design, communication networks, scalable modeling and simulation, and theoretical computer science His email address is eidenben@lanl.gov.

SUNIL THULASIDASAN is a scientist in the Information Sciences group at the Los Alamos National Laboratory. He received his Masters Degree in Computer Science from the University of Southern California in 2001 and his Bachelors Degree in Computer Science and Engineering from the University of Kerala, India in 1998. He research interests are in the areas of machine learning, high performance computing, graph algorithms, parallel simulations, computational geometry, complex networks and Internet modeling. He has been active in the area of discrete-event simulation software development for a number of years. He can be reached at sunil@lanl.gov.