A Scalable Analytical Memory Model for CPU Performance Prediction

Gopinath Chennupati[1](✉), Nandakishore Santhi[1], Robert Bird[1],
Sunil Thulasidasan[1], Abdel-Hameed A. Badawy[2], Satyajayant Misra[3],
and Stephan Eidenbenz[1]

[1] Los Alamos National Laboratory, SM 30, Los Alamos, NM 87545, USA
{gchennupati,nsanthi,bird,sunil,eidenben}@lanl.gov
[2] Klipsch School of Electrical and Computer Engineering,
New Mexico State University, Las Cruces, NM 88003, USA
badawy@nmsu.edu
[3] Computer Science Department, New Mexico State University,
Las Cruces, NM 88003, USA
misra@cs.nmsu.edu

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 including memory hierarchies, instruction pipelining, and speculative execution. We present Analytical Memory Model (AMM), a model of cache hierarchies, embedded in the Performance Prediction Toolkit (PPT) – a suite of discrete-event-simulation-based co-design hardware and software models. AMM enables PPT to significantly improve the quality of its runtime predictions of scientific codes.

AMM uses a computationally efficient, stochastic method to predict the reuse distance profiles, 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 profiles measured from the memory traces of applications on small instances. The analytical reuse profile is useful to estimate the effective latency and throughput of memory access, which in turn are used to predict the overall runtime of an application.

Our experimental results demonstrate the scalability of AMM, where we report the error-rates of three benchmarks on two different hardware models.

Keywords: Performance modeling · Cache hierarchies
Reuse distance · Probabilistic models · LLVM · Basic blocks

1 Introduction

The US DOE's exascale initiative demands a thousand-fold increase in supercomputing performance to meet the national needs in science, energy, and security.

S. Jarvis et al. (Eds.): PMBS 2017, LNCS 10724, pp. 114–135, 2018.
https://doi.org/10.1007/978-3-319-72971-8_6

The transition to exascale computing poses hard challenges in the form of design of future architectures. Moreover, confining to modulate either of the software or hardware is insufficient to meet the design goals. Co-design helps to trade-off the hardware designs and code development. Most of the research in co-design has been aimed at getting cycle accurate simulations in exploring the design space. Recent developments encourage novel performance modeling frameworks due to the black-box nature of the cycle accurate simulators [3]. Especially, cycle accurate simulators are slow and hinder the factors that contribute to the design of processors. Apart from the speed/slowness, many of these simulators are old while the modern processors are far more advanced than many of those models. Furthermore, the validation of these simulators is not as exhaustive as it should be, yet they are accepted in the research community. With that motivation, rapid performance prediction of computational codes on potential hardware architectures is a crucial requirement for pushing forward towards the exascale era.

In co-tuning the hardware and software parameters for physics codes, we introduce a novel framework, *Analytical Memory Model* (AMM), to explore the design space. AMM contains a compiler-driven static analysis of applications and a hardware-driven performance model. The compiler-driven analysis identifies the basic blocks (contain no loops and branches with a single entry and exit points) of a program, for which, an off-line analysis calculates the exact probability of executing a basic block. The hardware model is unique among the family of exascale co-design models with its capability to scale while considering the hardware specific factors such as frequency, latency, throughput, and cache. For the execution time, we consider the reuse distance [24] (the number of unique memory references between two references to the same addresses) and the number of CPU operations. We measure the total execution time of CPU operations using the pre-calculated instruction latencies.

In measuring the memory access time, we estimate a distribution of reuse distances from the memory trace of an application at a smaller input size. We randomly sample for each basic block and measure the conditional reuse distance profiles. These profiles together with the probability of executing a basic block results in the overall reuse profile of a program. The resultant reuse profiles help us estimate the availability of data (conditional hit rates) for a processor through various cache hierarchies. With the hit-rates, we measure the effective latency and throughput per memory operation. With the latency and throughput at hand, we measure the total memory access time of a program. The predicted runtime of an application is the sum of the time required for CPU operations and the total memory access time.

We evaluate AMM on three benchmarks: STREAM [23], Matrix Multiplication [15], and BlackScholes [6], on two hardware models – Intel Xeon and Intel Core i7. The results show that the sampled reuse profiles are similar to the real profiles, while the characteristic behavior of predicted runtimes is similar to actual runtimes on all benchmarks. Using the predicted runtimes, AMM offers insights into the optimal combination of hardware models for software applications when run in serial mode.

The rest of the paper is organized as follows: Sect. 2 presents the background; Sect. 3 describes AMM, Sect. 4 shows the experiments and the results; Sect. 6 concludes and recommends future research.

2 Background

2.1 Performance Modeling

Although the question, *How much execution time and energy does my algorithm cost?* [10] is not entirely new, but it helps to justify the trade-offs of the design decisions (time, energy, power, throughput, and latency). Since performance modeling with cycle-accurate simulations is too slow and cannot scale to large core counts, the framework in [34] introduced scalable performance prediction on the then HPC systems. Their prediction contains the simulation of an interconnect and a single processor performance, but unfortunately that does not scale on modern HPC machines.

Bailey and Snavely [4] developed an approach for performance prediction, which helps the stakeholders (system designers, co-design centers, and computational scientists) to improve the performance of applications.

For an optimal design decision, İpek *et al.* [18] explored the design space using neural networks, where they devised a non-linear regression model for which the data points in the design space are sampled at regular intervals. A machine learning framework, VERITAS [19], used sparse coding [27], that identified the performance characteristics (efficiency and resource significance) of proxy applications on a node. VERITAS compared the performance of proxy and real codes, which identified the factors that contribute to loss of efficiency. Another machine learning attempt [20] employed decision-trees on communication data and network hardware counters. These trees derived a strong correlation among a set of network features that contribute to the runtime.

In contrast, AMM accounts for factors such as memory hierarchy, processor latency, and throughput. Our model is intertwined with the Performance Prediction Toolkit (PPT) in predicting the runtimes of physics codes.

Structural Simulation Toolkit (SST) [29], a complex code execution simulator, offers some similar functionality but with different goals; unlike Performance Prediction Toolkit (PPT), relies on replicating control flow (*i.e.,* dynamically executes the application), models messaging behavior, scalable unlike cycle-accurate simulators.

2.2 Performance Prediction Toolkit

Performance Prediction Toolkit (PPT) developed at Los Alamos National Laboratory (LANL), is a scalable co-design framework, that has parameterized hardware and middleware models, accepts stylized codes as input and predicts the runtimes. PPT relies on Simian [31], a parallel discrete event simulation engine written in Python, Lua, and JavaScript. In Simian, each computing unit (host, compute node, CPU core) is an entity. Processes perform their tasks through message exchanges to remain active, sleep, wakeup, begin, and end. Simian advances the simulated time through a *time_compute*() function, that takes a *task list* – the number of CPU operations, memory usage, *etc.* The parameterized models of PPT use the task list to approximate the runtime. The hardware models – interconnects, compute nodes and CPU cores – mimic the lower level hardware processes using regression models resulting from PAPI [8] counters data.

The drawbacks of current PPT models are – regression often relies on inaccurate PAPI data; and the dependence on application developers expertise to explicitly specify the hit-rates. Alternatively, AMM predicts the hit-rates for a given input using an analytical reuse profile, we discuss the state-of-the-art in reuse distance calculation.

2.3 Reuse Distance

The reuse distance of a memory reference (M) is the number of distinct addresses in the trace after the most recent access to M. Memory traces were explored in a number of facets, including performance counters, reuse analysis, and cache behavior [26,33,35]. Our work differs in that, it improves concepts of *in-situ* reuse analysis from a memory trace. The reuse distances are used in defining a *reuse profile*, which is a distribution of reuse distances, that helps to estimate the availability of data in cache.

The compiler generated trace files for most scientific applications are often in tens and/or hundreds of gigabytes. Calculating reuse profiles from such large files is infeasible, moreover, the applications spend enormous amount of computational effort in generating these memory traces. Alternatively, synthetic traces [13] are used to estimate the reuse distributions. Partial Markov Model (PMM) [1] produced random memory references that rely on the existence of original trace and reported inaccuracies in the reuse profiles. Synthetic traces in [13] identified patterns in the memory references based on an analysis of instruction profiling, branches and dependencies. Attempts in [16] adapted least recently used stack models [7] over PMM states to accurately produce synthetic traces, their reuse profiles are accurate but unscalable.

Other attempts that sampled reuse profiles to study data locality include, StatCache [5], presented a probabilistic model that employs sampling to analyze the data locality on realistic workloads. Another sampling and parallelization attempt in [32] accelerated the reuse distance analysis on multi-cores. Unlike, these sampling attempts we use the memory trace of a single run of a program at smaller input size to estimate the reuse profiles at larger inputs. A recent approach [11] presented an analytical model to predict the performance and the energy consumption of a processor using architecture independent characteristics.

Of the attempts to approximate the reuse distance, Ding and Zhong [12] estimated the reuse patterns of a whole program based on training runs of a few small inputs. The model uses dependency analysis to estimate the cache misses with poor accuracy. In a different attempt, Chatterjee *et al.* [9] applied a set of formulas to characterize the cache misses, which perfectly handles nested loops and non-linear array layouts. Their model lacks the runtime knowledge of loop bounds. Sahoo *et al.* [30] tried to accurately characterize the cache miss count using reuse distances in the context of tensor contraction computations. Recently, reuse distance analysis predicted miss-rate per instruction [14], however, such a fine grained miss-rate estimation fail to scale.

In contrast to the existing attempts, AMM is simple, scalable and relies on Low-Level Virtual Machine (LLVM) [21] basic blocks (BB). We calculate reuse profiles for each BB of a program. These profiles are used to measure the cache hit-rates at different levels, which are used in predicting the runtimes of scientific applications.

3 Analytical Memory Model

AMM is a parameterized model for performance prediction, the factors that we consider in the prediction are: reuse distance distribution, latency and throughput of a program. The reuse profile corresponds to modeling different cache hierarchies of a processor in an elegant and scalable manner. These reuse profiles are used in estimating the availability of data from main memory to the processor via different cache levels. Further, we use data availability in calculating the latency and throughput of a program.

Figure 1 shows different steps of AMM in predicting the runtime of a program. AMM accepts a computer program (written in FORTRAN or C/C++) as

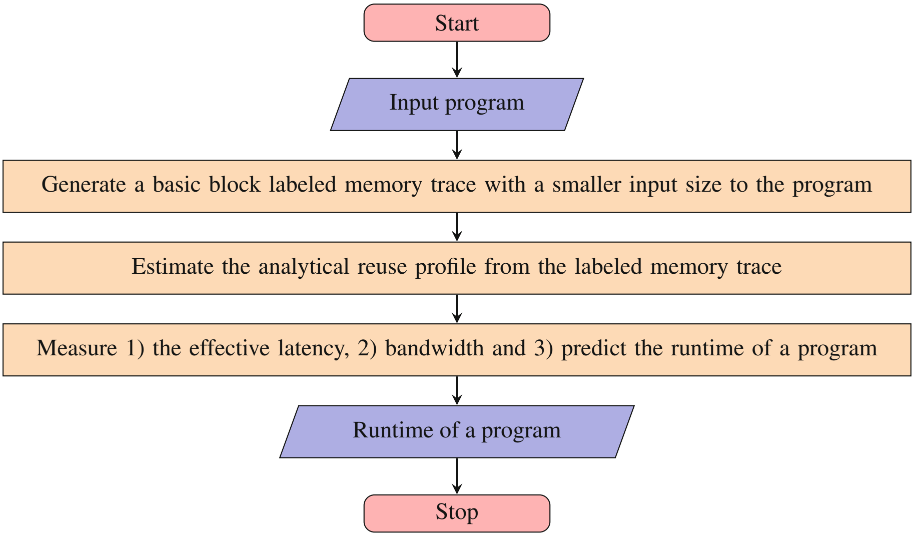
Start
Input program
Generate a basic block labeled memory trace with a smaller input size to the program
Estimate the analytical reuse profile from the labeled memory trace
Measure 1) the effective latency, 2) bandwidth and 3) predict the runtime of a program
Runtime of a program
Stop

Fig. 1. Different steps in analytical memory model (AMM)

an input, which is transformed into an intermediate representation (IR) using the compilation framework, LLVM. The transformation and analysis process involves: *(a)* generating a memory trace with basic block labels produced with a smaller input size of a program, *(b)* estimating the analytical reuse profiles of a program from the labeled memory trace, and *(c)* measuring the effective latency and throughput, with which, program runtime prediction can be made. We describe each step in detail as follows.

3.1 Generate Memory Trace

The first step in AMM is to generate a memory trace that contains the LLVM basic blocks. When the source code is compiled to produce IR, the transformed code consists of basic blocks. A basic block is a straight-line code with single entry and exit, with no intermediate branches except a branch at the exit.

The basic block labels in the trace of a program are generated using an LLVM characterization tool, Byfl [28], developed at LANL. We extended Byfl to instrument the memory addresses with LLVM basic block names. Note that LLVM does not create a distinct basic block for the function calls. We resolve such an ineptitude through preprocessing the labeled trace, where we ensure to distinguish the function calls as a separate basic block. For example, the i^{th} basic block (BB_i) of the labeled trace contains all the memory addresses that are generated as a result of executing the corresponding straight-line code of BB_i. Similar traces can be generated with Valgrind [25] and Pin [22], however, we use Byfl as it is developed using LLVM infrastructure. Like AMM, the attempts in [11] present a similar architecture independent performance and energy modeling.

3.2 Estimate Reuse Profile of a Program

The second step is to analytically estimate the reuse profile of a program ($Pr(D)$). The traditional methods of measuring the reuse profile are expensive due to large memory traces. Our technique promises to produce scalable memory traces at smaller inputs of a program, with which we estimate the reuse profiles at larger inputs. With the memory trace using smaller inputs, we estimate the reuse profile of a program as in Eq. 1

$$Pr(D) = \sum_{i=0}^{n(BB)} P(BB_i) \times P(D \mid BB_i) \tag{1}$$

where, D is the reuse distance, $n(BB)$ is the number of basic blocks, $P(BB_i)$ is the apriori probability of executing a basic block and $P(D \mid BB_i)$ is the conditional reuse profile of i^{th} basic block.

Algorithm 1 measures the conditional reuse profile of a basic block, BB_i. The algorithm takes the labeled trace as input, identifies all the instances of BB_i, from which, randomly select *sample_size* number of occurrences. For example, if a basic block appears hundred times in the trace, we randomly select $n\%$

Algorithm 1. Calculating the conditional reuse profile of a basic block (BB_i)

```
procedure $reuse\_profile\_BB_i(BB_i, memory\_trace)$
    $reuse\_distances, sampled\_wins \leftarrow [\,], [\,]$
    $sample\_size \leftarrow x$                              ▷ $x$% of all the $BB_i$(s)
    for $bb$ in $all\_BB_i$ do
        $sampled\_wins.append([BB_i\_start, BB_i\_end])$
    end for
    $windows \leftarrow random(sampled\_wins, sample\_size)$
    for $window$ in $windows$ do
        $reuse\_dist \leftarrow get\_rd(window, memory\_trace)$
        $reuse\_distances.append(reuse\_dist)$
    end for
    $uniq\_reuse\_dist, counts \leftarrow \text{unique}(reuse\_distances)$
    $prob\_rd \leftarrow$ map(lambda $x$: $x/len(reuse\_distances)$, $counts$)
    $r\_prof_i \leftarrow$ zip($uniq\_reuse\_dist$, $prob\_rd$)
    return $r\_prof_i$
end procedure
```

(typically 1%) of the samples from these occurrences. In fact, the reuse distance distributions are random due to uncertain memory mapping of program data. Therefore, it is important to randomly sample the trace, we term these random samples as *windows*. A window is a list that contains the start and the end indices of a sampled BB. We measure the reuse distances of all the memory addresses in a window, from which, calculate the corresponding probabilities.

Algorithm 2. Calculate the reuse distances

```
procedure $get\_rdist$(window, $memory\_trace$)
    $reuse\_dist \leftarrow [\,]$
    for idx, addr in enumerate($window$) do
        $window\_trace \leftarrow memory\_trace[:idx]$; $dict\_rd \leftarrow \{\,\}$; $addr\_found \leftarrow$ False
        for $addr\_idx$ in range(len($window\_trace$)) do
            $w\_addr \leftarrow window\_trace[-addr\_idx - 1]$
            if $addr == w\_addr$ then $addr\_found \leftarrow$ True;   break
            end if
            $dict\_rd[w\_addr] =$ True
        end for
        if $addr\_found$ then $reuse\_dist.append$(len($dict\_rd$))
        else $reuse\_dist.append(-1)$
        end if
    end for
    return $reuse\_dist$
end procedure
```

Algorithm 2 calculates the reuse distances memory addresses in a window. For each address in a window, we refer back in the trace from the current address to the exact same address, termed as *max back reference*. Once we find the memory address at two different indexes, the reuse distance for that address is the *cardinality* of the unique addresses between the two indexes. If the second index is absent, the reuse distance is infinite (∞). Similarly, the algorithm continues to measure the reuse distances for all the addresses in a basic block through a search for a *max back reference* in the original trace. At the end, the algorithm returns a list of all the reuse distances for that window.

Algorithms 1 and 2 calculate the reuse distances for all the addresses from all the sampled windows. Finally, we measure the frequency of each reuse distance, where the frequencies produce the respective probabilities. The reuse distances together with the corresponding probabilities form part of the *conditional reuse profile* of BB_i, $P(D \mid BB_i)$. The conditional reuse profiles are application dependent, for example, the conditional profiles of some applications may shift with input size. We extrapolate (see Sect. 4) these changes in conditional reuse profiles using polynomial regression techniques. Similarly, $P(BB_i)$ varies with the input size, measured as follows.

Measure $P(BB_i)$**:** Let us consider, BB_1, BB_2, ..., BB_j, ..., BB_{n-1}, BB_n is a series of basic blocks, any BB can execute any other BB. For example, the basic blocks BB_1, BB_2, ..., BB_k can execute BB_j, where, BB_1, ..., BB_k are termed as the predecessors of BB_j. Therefore, the predecessor BBs satisfy the following linear recursive relation:

$$N_j = \sum_{i \in Pred(j)} \pi_{ij} \times N_i \tag{2}$$

where, π_{ij} is the transition probability (measured off-line using compiler coverage analysis/application developer can identify manually) from predecessor block BB_i to BB_j. N_j is a homogeneous system of linear equations with many solutions. Since the entry basic block of most of the source codes is executed once, N_1 becomes 1.

Given π_{ij}, the apriori probability of a basic block ($P(BB_i)$) is defined as in Eq. 3:

$$P(BB_i) = \frac{N_i}{\sum\limits_{k=0}^{n(BB)} N_k} \tag{3}$$

where, N_i and N_k are the number of calls to the i^{th} and k^{th} basic blocks respectively.

$P(BB_i)$ changes with respect to the input size, however, we use the same labeled memory trace at smaller inputs to estimate the reuse profiles for larger instances of the program. We repeat our off-line analysis on $P(BB_i)$ in order to generate the apriori probabilities of basic blocks at bigger inputs. Note, the basic blocks with no memory access in their trace has no contribution towards the final reuse distribution.

3.3 Predict Runtime

The final step in AMM is to predict the runtime of an application. In runtime prediction, we measure latency and throughput using the reuse profile. The reuse profile calculates the availability of the data (hit-rates) from main memory to processor via different cache levels. The total predicted runtime of a program is the sum of the average memory access time (T_{avg_mem}) and the average time taken for the CPU operations (T_{CPU_ops}). The application characterization tool, Byfl is useful in counting the total memory required for the program and the number of CPU operations.

Therefore, the predicted runtime is measured with Eq. 4:

$$T_{pred} = T_{avg\_mem} + T_{CPU\_ops} \tag{4}$$

Probability of a Cache-Hit: In predicting the runtime, identifying the data availability at different cache levels is essential. With the analytical reuse profiles ($Pr(D)$), we measure the cache hit-rates (data availability) employing a *stack distance based cache model* (SDCM) [7], which helps to estimate the probability of a hit at any cache hierarchy (L_1, L_2, or L_3) for a given memory reference with a specific reuse distance. The following formula represents the probability of a hit for an *n-way* associative cache at a given reuse distance ($P(h \mid D)$):

$$P(h \mid D) = \sum_{a=0}^{A-1} \binom{D}{a} \left(\frac{A}{B}\right)^{a} \left(\frac{B-A}{B}\right)^{(D-a)} \tag{5}$$

where D is the 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 $B = 1024$ blocks. For a direct-mapped cache, $P(h \mid D)$ is $((B-1)/B)^D$ [7]. Therefore, the unconditional probability of a hit *P(h)* for the entire program can be approximated as in Eq. 6

$$P(h) = \sum_{i=0}^{N} P(D_i) \times P(h \mid D_i) \tag{6}$$

where, $P(D_i)$ is the probability of i^{th} reuse distance (D) in a reuse distribution $Pr(D)$. Herein, we investigate two variations (contiguous and non-contiguous) of runtime prediction with respect to the availability of data on memory and/or cache.

Case 1 (Contiguous): Memory Runtime Prediction. Assuming the contiguous availability of memory, the average memory access time is measured as in Eq. 7:

$$T_{avg\_mem} = \frac{\lambda_{avg} + (b-1) \times \beta_{avg}}{b} \times total\_mem \tag{7}$$

where λ_{avg} is average latency, β_{avg} is average reciprocal throughput, b is block size and *total_mem* is the total memory required by the program. The latency and throughput are per memory access, while the block size is considered as word size with the assumption of the availability of contiguous memory. Dividing the first term with block size will result in the average memory access time per byte, multiplying with *total_mem* results in the total memory access time of a program.

The hit-rates (Eq. 6) at different cache levels estimate the average latency and throughput of a given program. The average latency for a three-level cache is in Eq. 8

$$\lambda_{avg} = P_{L_1}(h) \times \lambda_{L_1} + (1 - P_{L_1}(h)) \Big[P_{L_2}(h) \times \lambda_{L_2} + (1 - P_{L_2}(h)) \big[P_{L_3}(h) \times \lambda_{L_3} + (1 - P_{L_3}(h)) \times \lambda_{RAM} \big] \Big] \tag{8}$$

where, λ_{L_1}, λ_{L_2}, λ_{L_3} and λ_{RAM} are the hardware specific measured latencies of L_1, L_2, L_3 caches and RAM respectively; $P_{L_1}(h)$, $P_{L_2}(h)$ and $P_{L_3}(h)$ are the probabilities of a hit for L_1, L_2 and L_3 caches respectively, that are calculated using Eq. 6. Similarly, we measure the average throughput, β_{avg} (replace λs in Eq. 8 with β).

Case 1 (Contiguous): Measure. T_{CPU_ops} Byfl and/or a simple off-line analysis helps to identify the number of CPU operations (ADD, SUB, and DIV, etc.) of a program. We measure the time required for CPU operations using the hardware specific instruction latencies and the operations count, thus, the total runtime is predicted as T_{pred} (Eq. 4).

Case 2 (Non-contiguous): Memory Runtime Prediction. In measuring the average memory access time, as opposed to the previous consideration, we consider the non-contiguous alignment of memory, as is the case in reality. There will be gaps (v) in between the required program data, therefore, the new block size (b in Eq. 7 becomes b^{new}): $b^{new} = b + v$. However, the entire block may not always be transferred from main memory to different cache levels due to the dependence on factors such as data bus width, and cache size, etc. Therefore, we model such a unique behavior of cache as follows. Let us consider, b_1^{new}, b_2^{new}, b_3^{new}, ..., b_i^{new}, ..., b_n^{new} are the blocks of data on main memory, while C be the amount of data transferred on to a cache from main memory at any given time. Thus, the new block size at a given cache size (B) can be re-written as:

$$b^{new} = \begin{cases} C & : \textbf{if} \quad b_i^{new} \leq C \\ \left\lceil \dfrac{b_i^{new}}{C} \right\rceil \times C & : \textbf{if} \quad B \geq b_i^{new} \geq C \\ B & : \textbf{if} \quad b_i^{new} \geq B \end{cases}$$

Case 2 (Non-contiguous): Time for CPU Operations (T_{CPU_ops}). In case of the time taken for CPU operations, there is a large difference in the instruction latencies between DIV and the rest of the instructions. Moreover, the time required for CPU operations is dependent on program characteristics, where some applications are instruction latency dependent while others are throughput reliable. Thus, the time for the resultant CPU operations is:

$$T_{CPU\_ops} = \begin{cases} \lambda_{in} + (N_{in} - N_{in\_div} - 1) \times \beta_{in} + \lambda_{div} + (N_{in\_div} - 1) \times \beta_{div} & : \textit{throughput} \\ (N_{in} - N_{in\_div} - 1) \times \lambda_{in} + (N_{in\_div} - 1) \times \lambda_{div} & : \textit{latency} \end{cases}$$

where λ_{in}, λ_{div}, β_{in} and β_{div} are latencies and throughputs of instructions, ADD/SUB, MUL and DIV respectively, while N_{in} and N_{in_div} are the number of instructions.

4 Experiments

In this section, we describe the target architectures and the benchmark applications used in validating our model.

4.1 Target Architectures

We use AMM to validate three different benchmark applications on two hardware architectures. Table 1 presents the two processor architectures, each of which uses three cache levels with different sizes. The L_3 cache of Intel Xeon processor is shared among the available cores on the chip while that of the Intel Core i7 is unshared.

In predicting the runtimes, we build the hardware models for the two experimental processors (Table 1) along with AMM in Performance Prediction Toolkit (PPT). PPT has parametrized hardware models and software proxy applications. The hardware parameters of PPT are: cache latencies, cache sizes, cache line sizes, associativity, and memory bandwidth (throughput) at different cache levels (we consider the reciprocal throughput), RAM latency, and data bus width. The hardware parameters are measured values for a given processor, reasonably reliable sources include Agner Fog's manual [2], Intel and others[1] present these parameter values for a number of hardware architectures. We can measure

Table 1. The target architectures and their parameters

#	Processor	Speed	Cache size (bytes)			Shared
		(GHz)	L_1	L_2	L_3	L3?
1	Intel Xeon E5-2695	2.10	64K	256K	45M	Yes
2	Intel Core i7-4770HQ	2.20	256K	1M	6M	No

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

these parameters using standard benchmarks, nevertheless, the objective in this paper is performance modeling rather parameter calibrations. The latencies and throughputs used in the hardware model include both at the cache hierarchies and the instructions such as *ADD/SUB*, *MUL*, and *DIV*. The software parameters are: total memory of an application, the number of integer and floating point operations (add, mul, etc.), and the block size (Eq. 7 in Sect. 3.3), measured using Byfl.

4.2 Benchmarks

The three benchmark applications we used are: STREAM [23], matrix-matrix multiplication (MM) [15], and BlackScholes [6].

STREAM is a memory benchmark with vectors of floating point operations. STREAM contains four kernels: *ADD* performs the sum of two vectors; *SCALE* multiplies a vector with a floating-point scalar; *COPY* assigns one vector into another and *TRIAD* performs the above three operations. We execute all the above four kernels.

MM is a naive implementation (*ijk* method that has 3 nested loops) of floating-point matrix-matrix multiplication. MM, in this paper, is defined as $R = \alpha P \times Q + \beta R$, where P, Q and R are m × k, k × n and m × n matrices respectively while α and β are floating-point scalars.

BlackScholes is a PARSEC benchmark, partial differential model used to predict the European stock option prices. BlackScholes functions within two nested loops, where the outer-loop stands for the number of iterations of the algorithm and the inner loop performs the floating-point operations needed for option prices.

5 Results

We implemented the respective proxy application in PPT for all the three benchmarks. We validate AMM for these three applications as follows: (1) compare the real and predicted reuse profiles, and (2) compare the real and predicted runtimes. Both the simulation and actual runs are computed on a single core of a CPU.

5.1 Validate Reuse Profile

Our goal is to validate the analytical reuse profiles with that of the actual profiles. The reuse profiles are discrete, in general, they are architecture independent due to which the reuse profiles are same across the two experimental hardware architectures.

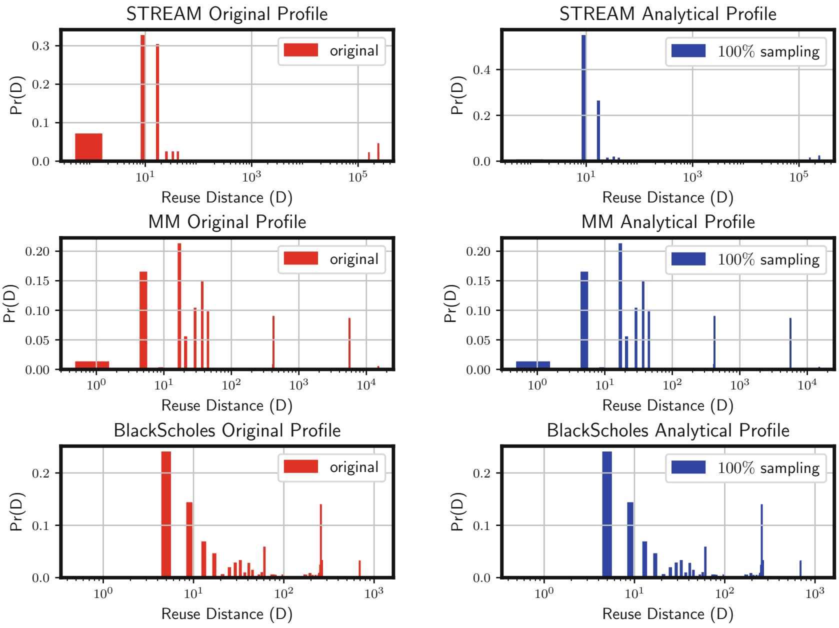

Fig. 2. Compare the original (left) and analytical (right) reuse profiles of STREAM (top), MM (middle), and BlackScholes (bottom) at input sizes of 10000 floating-points, matrix of size 25 × 25 and 16 data points respectively. Original distribution is measured using a stack based algorithm, while that of the analytical is measured using AMM with 100% sampling. The reuse distance (D) is in *log* scale while Pr(D) is in decimal scale.

Figure 2 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 the probability ($P(BB_i) \times P(D|BB_i)$) of execution of that basic block. We adopted 100% sampling in order to validate the actual and analytical profiles, in the runtime prediction, we consider 1% sampling, which guarantees scalability. On all the three benchmarks, AMM calculated reuse distances (D, on X-axis) are identical to that of the actual reuse distances, so does their number of occurrences. The corresponding probabilities (Pr(D), on Y-axis) are approximately similar, the analytical probabilities are slightly higher at a few reuse distances because of their dependence on the accuracy of $P(BB_i)$. Nonetheless, these inaccuracies have insignificant impact on the final cache hit-rate, therefore, the analytical reuse profiles are similar to the actual.

The original reuse profiles are measured using a stack [24] based implementation that has a time complexity of $O(NM)$. The analytical reuse profiles are measured using Algorithm 1, which has a computational complexity of $O(NSB)$ $\sim O(N)$, since the number of samples (S) and size of the basic block (B) are constant. The worst case complexity is $O(NM)$, in the case of 100% *sampling*, which will never happen.

5.2 Validate Runtime

We validate the AMM predicted runtimes with that of the actual for all the three benchmark applications at different input sizes on both the target architectures. Table 2 presents four different input sizes for each of the three benchmarks. For example, STREAM has three floating-point vectors, all of which are initialized with same input size. The inputs for each run of STREAM varies from 10000, 20000, 30000 to 40000 elements. Similarly, MM and BlackScholes have four square matrix sizes and four datasets (16, 32, 64, and 128 data-points) respectively. We report both the actual and predicted runtimes at four different input sizes on each benchmark.

Table 2. Benchmarks with different input sizes.

#	Program	Input sizes
1	STREAM	{10000, 20000, 30000, 40000}
2	MM	{25 × 25, 50 × 50, 100 × 100, 200 × 200}
3	BlackScholes	{16, 32, 64, 128}

In predicting the runtimes, we analytically estimate the reuse profiles at each input using the memory trace (1% sampling) for the smaller input size of the respective benchmark. For example, in the case of MM, we use the memory trace at an input size of 25 × 25 as the base to estimate the reuse profiles at 50 × 50, 100 × 100 and 200 × 200. The probabilities of basic blocks (P(BB_i)) change as the input size changes.

Figure 3 shows the analytical (1% sampling) reuse profiles of both the benchmarks at different input sizes. The sampled reuse profiles are approximately similar to that of the original, however, some large but relatively rare reuse distances disappear due to random sampling. For example, if a basic block occurrence appears at the bottom of the memory trace, there is a chance to omit such occurrences due to 1% random sampling, thereby, the larger reuse distances disappear. These large values may have significant impact on the cache hit-rates, thus, we propose to extrapolate these reuse distances, similar to Zhong et al. [36], where the prediction of program locality with respect to inputs identifies the data access patterns and builds a parametrized model for extrapolation. In contrast to Zhong et al., we extrapolate the conditional reuse distances of basic blocks (instead of the whole program) at larger input sizes of a program using the reuse distances at a few smaller inputs. In fact, extrapolating the conditional reuse profiles of basic blocks using small input reuse distances preserves our promise of scalable AMM.

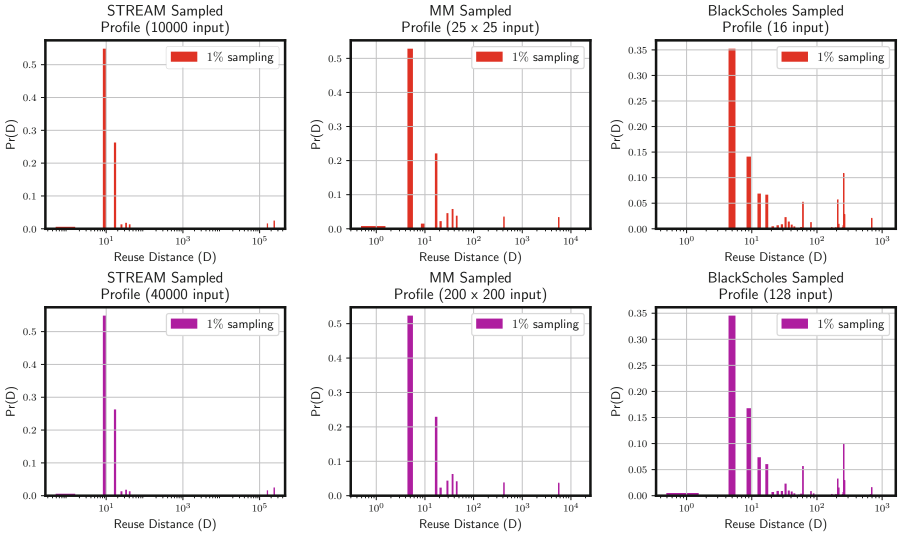

Fig. 3. Sampled analytical reuse profiles (reuse distance (D) is on *log scale*) of the three benchmarks: STREAM (left), matrix multiplication (middle), and BlackScholes (right) at different input sizes. The rate of sampling is 1%, while the base memory traces for the three benchmarks are at input sizes of 10000, 25 × 25, and 16 respectively.

Extrapolate the Conditional Reuse Distances of a BB. From the probability distribution of executing the basic blocks, we observe that a few number of the total basic blocks of a program have significant impact on the reuse profiles. Figure 4 (left) shows the probability of executing each basic block ($P(BB_i)$) at different small input sizes (10, 12, 15, 17, and 20) of MM. Of all the *twenty two* basic blocks of a MM program, $BB15 - BB17$ have relatively significant contribution over the remaining basic blocks. Empirically, the number of entries in the conditional reuse profiles of these three basic blocks grow with the input size, while that of the remaining basic blocks remain consistent irrespective of the input size. Therefore, extrapolating the conditional reuse distances of these significant BBs helps in identifying the missing large reuse distances.

We explain the extrapolation strategy on one of the three BBs, $BB15$, where we find that the first few (seven for MM) reuse distance entries of the distribution remain unchanged irrespective of the inputs. Probability of these reuse distances contribute 75% of the distribution, while the other growing reuse distances contribute the remaining 25%. Since these initial entries are consistent, what the following linear relation (Eq. 9) predicts is useful in estimating the reuse distances at any input size (x).

$$D_i^{'}|x = D_i \quad \forall\, i = 1 \dots 7 \tag{9}$$

where, $D_i^{'}|x$ is new reuse distance at an input, D_i is the reuse distance of a basic block.

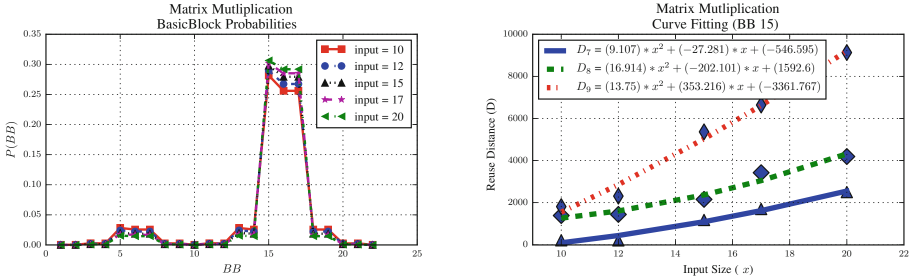

Fig. 4. Probabilities (left) of all basic blocks (BBs) of MM at multiple small inputs. Extrapolation of reuse distances (right) as a function of input size (x) for matrix multiplication using the data from five small runs at five different input sizes.

We extrapolate the remaining reuse distance entries that grow with the input size, where the number of these entries are inconsistent at each input size. In order to regulate these inconsistencies, we apply a *fixed-binning* strategy, in which, we use a constant number of bins, each of which represents an average of the subset-of-reuse-distances. The number of entries within a bin changes while the total number of bins remain same.

Figure 4 (right) shows the extrapolation of three bins using five small input sizes. The points represent the average reuse distances for each bin while the curves represent the predicted polynomial fit for each bin as a function of input size (x). Note, the input (x) in the extrapolated curves is on one-dimension of MM. We observe that the predicted average reuse distances grow in polynomial fashion. Similarly, n we can estimate the respective probabilities of these reuse distances, together forms the extrapolated conditional reuse profile of a BB. We can increase the number of bins, however, estimating the hit-rates relies on the magnitude of the reuse distances rather the distinct number of reuses alone. That way, we extrapolate the conditional reuse profiles of the most significant basic blocks of an application and combine the reuse profiles of all the basic blocks to produce the complete reuse profile of a program.

Our prediction strategy does not incur the extra computational overhead of extrapolating the reuse profiles on the whole program (as opposed to Zhong et al., therefore, AMM is scalable), while approximates the hit-rates with reasonably good accuracy.

Is the Data Available for Use by the Processor? Given the reuse profiles, it is essential to analyze the availability of data for the processor. 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 the input sizes of 10000, 25 × 25 and 16 of STREAM, MM and BlackScholes respectively. Since the reuse distances are independent of the underlying hardware, we use the same reuse profile (with respect to the benchmark) to measure the cache hit-rates at different cache sizes. However, the conditional hit-rates at a given stack distance are calculated on *Intel Xeon E5-2695* architecture. The reuse distance (D) is on a *log scale*, whereas B_1, B_2 and B_3 are cache sizes measured in terms of the number of blocks (cache-size/cache-line-size). $P_{L_1}(h|D)$, $P_{L_2}(h|D)$ and $P_{L_3}(h|D)$ are conditional hit-rates at three cache levels L_1, L_2 and L_3 respectively. On all the benchmarks, the cache hit-rate at a reuse distance ($P_{L_1}(h|D)$) suddenly drops for L_1 cache after the cache size (B_1), which confirms that the application data exceeds the L_1 cache of Intel Xeon processor. A similar behavior is found on L_2 cache in the case of STREAM and matrix multiplication, while for BlackScholes the data exists on L_2 cache. STREAM data slightly exceeds the L_3 limits, while the data of the remaining two benchmarks is available on L_3. We found that the probability of the corresponding large reuse distances ($P(D_i)$ in Fig. 3) is approximately zero.

However, for *Intel Core i7-4470HQ* – BlackScholes data exists in L_1 cache and the remaining two benchmarks data does not exist; on L_2, STREAM data is not present while the remaining two benchmarks data does exist; L_3 can hold the data for all the three benchmarks. Since Intel Core i7 has relatively large L_1 and L_2 cache sizes, the data is readily available for the processor. Intel Core i7 have relatively smaller L_3 cache size compared with that of Intel Xeon. Intel Core i7 processors L_3 capacity is insufficient for large input sizes of a program. In addition, L_3 cache of Intel Xeon is shared among the available cores while that is not the case with Core i7. With these characteristics, reuse distances that exceed the cache sizes are always a miss. These observations (Fig. 5) suggest that the availability of data in the cache depends on the target architectures and the application data requirements.

A discussion on the locality of data is out of the scope. However, this study shows that the 1% sampled reuse profiles are reasonably better approximations in estimating the runtime of an application. Therefore, for better availability of data, we suggest to design a processor with the L_1 and L_2 caches of Intel Core i7 and the L_3 of Intel Xeon.

Prediction of Run-Times: We validate the predicted runtimes, Fig. 6 presents the error-rates of the AMM predicted runtimes when compared wxith that of the actual for all the three benchmarks at different input sizes on the two target architectures. We assume that the processor executes one application at a given time, so that the cache and RAM are available for the application. We observe that Intel Core i7 error-rates of STREAM are significantly higher than the Intel Xeon due to small L_3 cache size of Core i7. For the remaining two benchmarks (MM and BlackScholes), the difference in the predicted error-rates across both the target architectures is insignificant, The reason being the fact that the application data for these two benchmarks fits in the cache hierarchy.

We observe that AMM over-predicts the runtimes when compared to the actual runtimes, especially, at larger input sizes. Although we over-predict, the characteristic behavior of the runtimes with respect to the input remains in coherence with the actual runtimes. The reason behind the over-prediction is due to the fact that AMM is purely a *memory model*. We can reduce such over-prediction through a model for pipelines along with the memory model. AMM

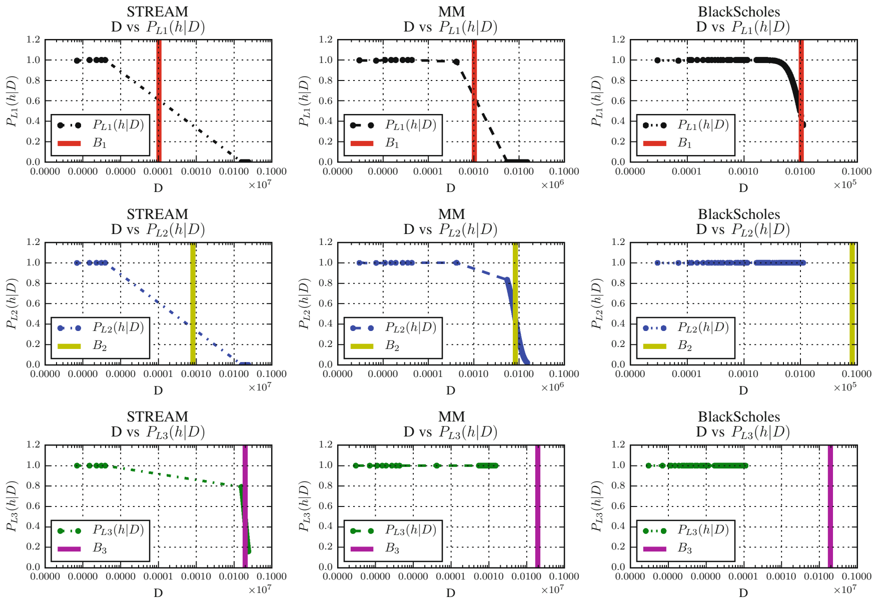

Fig. 5. Conditional cache hit-rates at a given reuse distance for all the three benchmarks – STREAM (left), matrix multiplication (middle) and BlackScholes (right). B_1, B_2 and B_3 are the cache sizes in terms of number of blocks (see Sect. 3.2) for L_1, L_2 and L_3 caches respectively.

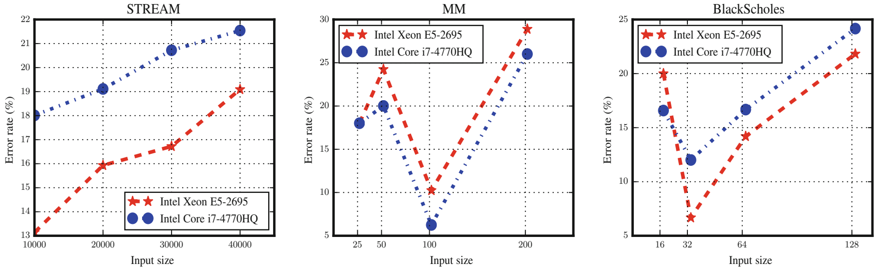

Fig. 6. Error-rates of predicted runtimes (with respect to the actual runtimes) for the three benchmark applications (STREAM, matrix multiplication (MM) and BlackScholes) on both the target architectures (Intel Xeon E5-2695, Intel Core i7-4770HQ).

assumes the execution of the program in complete sequential mode, whereas the actual CPU core executes the independent tasks simultaneously through pipelines. In addition to pipelines, factors such as prefetching, replacement strategy, TLB, vector operations, micro-architecture, and etc. can have higher dividends in performance prediction. Building a parameterized model (one of our future directions to investigate) using these factors that work hand-in-hand with AMM would reduce the over-prediction in runtimes. Although the pipeline effect is not present in AMM, the predicted runtimes are reasonably abreast with that of the actual while we also claim that the characteristic behavior of the predicted runtimes is akin to the actual runtimes of all the benchmarks on both the target architectures.

Between Intel Xeon and Core i7, the latter is much faster than the former on these set of benchmarks due to higher clock speed. Our observations (Fig. 5) in cache sizes play a significant role in making the data available for processor, which obviously impacts the performance of an application. Intel Core i7 clearly has larger L_1, L_2 caches and a smaller L_3 cache, which in fact, is insufficient for large applications that might have adverse effects on performance despite processor speed. Intel states that the *Broadwell* family of Xeon processors are less powerful and energy efficient compared to the *Haswell* of Intel Core i7. With our study, we believe that increasing the L_1 and L_2 cache sizes of Xeon processors might further boost the performance with little/minimum effect on energy consumption, especially, when the execution of an application becomes concurrent/parallel.

6 Conclusion

We presented a novel analytical memory model (AMM) that produces basic block labeled memory traces using LLVM instrumentation. The memory traces at smaller inputs are randomly sampled to produce the reuse distance distributions at larger inputs for scientific applications. Using the smaller input memory traces, reuse distance profiles of the applications are estimated at larger input sizes. The analytically measured reuse profiles are similar to the actual reuse profiles. Further, the estimated reuse profiles are used to predict the runtimes of the applications. Our hardware model consists of low-level details such as latency, throughput of different hardware components (cache levels, RAM, etc.) and CPU instructions (*add*, *sub*, *mul*, etc.). The runtime results are consistent with the real runtimes while the characteristic behavior of the predicted runtimes is similar to that of the actual runtimes. We observed that AMM over-predicted the runtimes due to nonexistence of pipeline, cache prefetching, hardware threads, and TLB in the hardware model. Developing and integrating these missing models would guarantee a close prediction, therefore, is one of our future directions. With the addition of pipelines in AMM, similar to [4,17,20], we aim to predict the performance of MPI aware applications. Nevertheless, having AMM like fine-grained hardware model is essential for accurate and scalable performance prediction in distributed environments.

References

1. Agarwal, A., Hennessy, J., Horowitz, M.: An analytical cache model. ACM Trans. Comput. Syst. **7**(2), 184–215 (1989)
2. Agner, F.: Instruction tables: lists of instruction latencies, throughputs and micro-operation breakdowns for intel, AMD and VIA CPUs. Technical University of Denmark, Copenhagen, Denmark (2016)

3. Austin, T., Larson, E., Ernst, D.: Simplescalar: an infrastructure for computer system modeling. Computer **35**(2), 59–67 (2002)
4. Bailey, D.H., Snavely, A.: Performance modeling: understanding the past and predicting the future. In: Cunha, J.C., Medeiros, P.D. (eds.) Euro-Par 2005. LNCS, vol. 3648, pp. 185–195. Springer, Heidelberg (2005). https://doi.org/10.1007/11549468_23
5. Berg, E., Hagersten, E.: StatCache: a probabilistic approach to efficient and accurate data locality analysis. IEEE Int. Symp. ISPASS Perform. Anal. Syst. Softw. **2004**, 20–27 (2004)
6. Bienia, C., Kumar, S., Singh, J.P., Li, K.: The parsec benchmark suite: characterization and architectural implications. In: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, PACT 2008, New York, NY, USA, pp. 72–81. ACM (2008)
7. Brehob, M., Enbody, R.: An analytical model of locality and caching. Technical report MSU-CSE-99-31 (1999)
8. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.: A portable programming interface for performance evaluation on modern processors. Int. J. High Perform. Comput. Appl. **14**(3), 189–204 (2000)
9. Chatterjee, S., Parker, E., Hanlon, P.J., Lebeck, A.R.: Exact analysis of the cache behavior of nested loops. In: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, PLDI 2001, New York, NY, USA, pp. 286–297. ACM (2001)
10. Choi, J.W., Vuduc, R.W.: How much (execution) time and energy does my algorithm cost? XRDS **19**(3), 49–51 (2013)
11. den Steen, S.V., Eyerman, S., Pestel, S.D., Mechri, M., Carlson, T.E., Black-Schaffer, D., Hagersten, E., Eeckhout, L.: Analytical processor performance and power modeling using micro-architecture independent characteristics. IEEE Trans. Comput. **65**(12), 3537–3551 (2016)
12. Ding, C., Zhong, Y.: Predicting whole-program locality through reuse distance analysis. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, PLDI 2003, pp. 245–257. ACM (2003)
13. Eeckhout, L., de Bosschere, K., Neefs, H.: Performance analysis through synthetic trace generation. In: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2000, Washington, DC, USA, pp. 1–6. IEEE (2000)
14. Fang, C., Carr, S., Önder, S., Wang, Z.: Reuse-distance-based miss-rate prediction on a per instruction basis. In: Proceedings of the 2004 Workshop on Memory System Performance, MSP 2004, New York, NY, USA, pp. 60–68. ACM (2004)
15. Gunnels, J.A., Henry, G.M., van de Geijn, R.A.: A family of high-performance matrix multiplication algorithms. In: Alexandrov, V.N., Dongarra, J.J., Juliano, B.A., Renner, R.S., Tan, C.J.K. (eds.) ICCS 2001. LNCS, vol. 2073, pp. 51–60. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45545-0_15
16. Hassan, R., Harris, A., Topham, N., Efthymiou, A.: Synthetic trace-driven simulation of cache memory. In: 21st International Conference on Advanced Information Networking and Applications Workshops, vol. 1 of AINAW 2007, pp. 764–771 (2007)
17. Ipek, E., de Supinski, B.R., Schulz, M., McKee, S.A.: An approach to performance prediction for parallel applications. In: Cunha, J.C., Medeiros, P.D. (eds.) Euro-Par 2005. LNCS, vol. 3648, pp. 196–205. Springer, Heidelberg (2005). https://doi.org/10.1007/11549468_24

18. Ipek, E., McKee, S.A., Caruana, R., de Supinski, B.R., Schulz, M.: Efficiently exploring architectural design spaces via predictive modeling. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XII, New York, NY, USA, pp. 195–206. ACM (2006)
19. Islam, T.Z., Thiagarajan, J.J., Bhatele, A., Schulz, M., Gamblin, T.: A machine learning framework for performance coverage analysis of proxy applications. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Piscataway, NJ, USA, pp. 46:1–46:12. IEEE (2016)
20. Jain, N., Bhatele, A., Robson, M.P., Gamblin, T., Kale, L.V.: Predicting application performance using supervised learning on communication features. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2013, New York, NY, USA, pp. 95:1–95:12. ACM (2013)
21. Lattner, C., Adve, V.: Llvm: a compilation framework for lifelong program analysis & transformation. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization, CGO 2004, Washington, DC, USA, pp. 75–87. IEEE (2004)
22. Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2005, New York, NY, USA, pp. 190–200. ACM (2005)
23. Luszczek, P.R., Bailey, D.H., Dongarra, J.J., Kepner, J., Lucas, R.F., Rabenseifner, R., Takahashi, D.: The hpc challenge (hpcc) benchmark suite. In: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, SC 2006, New York, NY, USA. ACM (2006)
24. Mattson, R.L., Gecsei, J., Slutz, D.R., Traiger, I.L.: Evaluation techniques for storage hierarchies. IBM Syst. J. **9**(2), 78–117 (1970)
25. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2007, New York, NY, USA, pp. 89–100. ACM (2007)
26. Nguyen, A.T., Bose, P., Ekanadham, K., Nanda, A., Michael, M.: Accuracy and speed-up of parallel trace-driven architectural simulation. In: Proceedings 11th International Parallel Processing Symposium, pp. 39–44. IEEE (1997)
27. Olshausen, B.A., Field, D.J.: Sparse coding with an overcomplete basis set: a strategy employed by v1? Vis. Res. **37**(23), 3311–3325 (1997)
28. Pakin, S., McCormick, p.: Hardware-independent application characterization. In: International Symposium on Workload Characterization (IISWC), Portland, Oregon, USA, pp. 111–112. IEEE (2013)
29. Rodrigues, A.F., Murphy, R.C., Kogge, P., Underwood, K.D.: The structural simulation toolkit: exploring novel architectures. In: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, SC 2006, New York, NY, USA, p. 157. ACM (2006)
30. Sahoo, S.K., Panuganti, R., Sadayappan, P., Krishnamoorthy, P.: Cache miss characterization and data locality optimization for imperfectly nested loops on shared memory multiprocessors. In: Proceeding of the 19th IEEE International Parallel and Distributed Processing Symposium, pp. 44–53 (2005)

31. Santhi, N., Eidenbenz, S., Liu, J.: The simian concept: parallel discrete event simulation with interpreted languages and just-in-time compilation. In: Proceedings of the 2015 Winter Simulation Conference (WSC), pp. 3013–3024. IEEE (2015)
32. Schuff, D.L., Kulkarni, M., Pai, V.S.: Accelerating multicore reuse distance analysis with sampling and parallelization. In: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, PACT 2010, New York, NY, USA, pp. 53–64. ACM (2010)
33. Sherwood, T., Perelman, E., Hamerly, G., Calder, B.: Automatically characterizing large scale program behavior. In: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS X, New York, NY, USA, pp. 45–57. ACM (2002)
34. Snavely, A., Carrington, L., Wolter, N., Labarta, J., Badia, R., Purkayastha, A.: A framework for performance modeling and prediction. In: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing, SC 2002, Los Alamitos, CA, USA, pp. 1–17. IEEE (2002)
35. Weinberg, J., McCracken, M.O., Strohmaier, E., Snavely, A.: Quantifying locality in the memory access patterns of hpc applications. In: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, SC 2005, Washington, DC, USA, pp. 50–61. IEEE (2005)
36. Zhong, Y., Shen, X., Ding, C.: Program locality analysis using reuse distance. ACM Trans. Program. Lang. Syst. **31**(6), 20:1–20:39 (2009)