A Markov-Model-Based Framework for Supporting Real-Time Generation of Synthetic Memory References Effectively and Efficiently

Alfredo Cuzzocrea
DIA Dept., University of Trieste and ICAR-CNR, Italy
alfredo.cuzzocrea@dia.units.it

Enzo Mumolo
DIA Dept., University of Trieste, Italy
mumolo@units.it

Marwan Hassani
MCS Dept., Eindhoven University of Technology, The Netherlands
m.hassani@tue.nl

Giorgio Mario Grasso
COSPECS Dept., University of Messina, Italy
gmgrasso@unime.it

Abstract

Driven by several real-life case studies and in-lab developments, synthetic memory reference generation has a long tradition in computer science research. The goal is that of reproducing the running of an arbitrary program, whose generated traces can later be used for simulations and experiments. In this paper we investigate this research context and provide principles and algorithms of a Markov-Model-based framework for supporting real-time generation of synthetic memory references effectively and efficiently. Specifically, our approach is based on a novel Machine Learning algorithm we called Hierarchical Hidden/non Hidden Markov Model *(HHnHMM). Experimental results conclude this paper.*.

1 Introduction

One of the problems with trace driven simulation is that trace collection and storage are time and space consuming procedures. To collect a trace, hardware or software monitors are used. The amount of data to be saved is of the order of hundreds or thousands of megabytes for some minutes the program executions. This is necessary to produce reliable results [21]. Due to the large amount of data to be processed the computer time is also very long. Several techniques have been proposed to reduce the cache simulation time: trace stripping, trace sampling, simulation of several cache configurations in one pass of the trace [35] and parallel simulation [19, 31]. Synthetic traces have been proposed as an alternative to secondary-storage based traces since they are faster and do not demand disk space. They are also attractive since and they could be controlled by a reduced set of parameters which regulate the workload behavior. The problem of Synthetic traces is that it is difficult to exactly mimic the real behavior of the addressed program, thus limiting the use of the traces to early evaluation stages. Many studies, for example [14, 36], have highlighted the difficulty to exactly describe original characteristics of the memory references, such as locality, with analytic models.

On the other hand, driven by several real-life case studies and in-lab developments, synthetic memory reference generation has a long tradition in computer science research, as confirmed by several recent research initiatives (e.g., [6, 5, 24]).

In this paper we use a machine learning approach for describing collected traces. In particular, a specific type of Markov Model (MM), the Hierarchical Hidden/non Hidden HHnMM, where each state of MM is linked to an HMM for producing sequences of labels, not just labels as in standard HMM, has been worked out. This approach is attractive because on one side the behavior of the execution is learned by the model to ensure by machine learning that the artificial sequence will mimic the behavior of the original execution and on the other side, making use of the generation characteristic of the Ergodic Hidden Markov Models, sequences of any lengths can be generated. The machine learning framework requires that a suitable feature representation of the executions is provided, as we will describe shortly. Our approach consists of a learning phase, where a real trace is analyzed with the aim to derive the features for training the HHnMM, and a generation phase, where a synthetic trace is generated from HHnMM. A preliminary version of this

DOI reference number: 10.18293/DMSVIVA2018-018

paper appears in the workshop paper [13].

2 Methodology Overview

For performance analysis of the memory subsystem of a new computer system, we may generate a long sequence of memory references from some given testing application. Generally this require to store the long sequences on a disk, which may occupy many gigabytes of disk space. This may lead to disk space unavailability or data transfer delay problems. The alternative approach is to artificially generate a sequence of memory references similar to that required by the same given software application. In Figure 1, we summarize the trace analysis algorithm described in this paper.

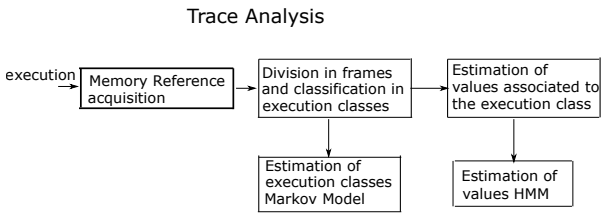

Figure 1. Trace analysis algorithm

First of all we must reduce the memory references produced by an application to a simpler representation. Thus, we divide the memory references sequence in frames, and each frame is classified as belonging to some execution classes, for example *Sequential* or *Periodic*. The execution classes are easily estimated from the reference traces. The sequence of memory references is thus transformed into a sequence of execution classes, which is a sequence with very few labels. This sequence is modeled with an Ergodic Markov Model, which is the Non Hidden part of the model. To each state of this MM, an Ergodic Hidden Markov Model is linked, for modeling the sequence of values associated with each execution class. For example the *Periodic* execution class is associated to the value of Period, or Loop width, which may change for each periodic frame. Once the non Hidden and the Hidden Markov Models are trained, artificial memory reference sequences can be generated by using the generation characteristic of the Markov Models. Namely, starting from an initial node of the MM which describe the execution classes, we generate a sequence of values associated to the execution class by visiting the associated HMM. The generation of synthetic memory references is summarized in Figure 2.

For example one may want to generate the memory references generated by the C compiler, *gcc*. The key of our algorithm is that the compiler produces somehow different memory references when used to compile different C

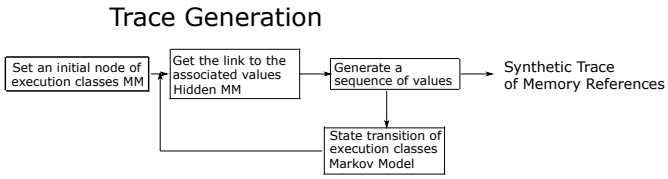

Figure 2. Synthetic trace generation algorithm

sources. The different memory references sequence anyway should contain a common structure because the same compiler *gcc* is used.

3 Describing Executions from Memory References

In this Section, we deal with the identification of the type of execution starting from the sequence of the memory reference patterns captured from the running programs. Memory reference patterns have been studied in the past by many authors with the goal to improve program execution on high performance computers or to improve memory performance. Tools to understand memory access patterns of memory operations performed by running programs are described also by Choudhury *et al.* in [8]. Such studies are directed towards the optimization of data intensive programs such as those found in scientific computing.

Other works, for example [16, 23, 27, 32], have the goal to improve memory performance, since memory systems are still the major performance and power bottleneck in computing systems. In particular, Harrison *et al.* describe in [16] the application of a simple classification of memory access patterns they developed earlier to data prefetch mechanism for pointer intensive and numerical applications. Lee *et al.* exploit the regular access patterns of multimedia applications to build hardware prefetching technique that is assisted by static analysis of data access pattern with stream caches.

Several papers by Choi *et al.*, namely [26, 7] , analyze streams of disk block requests. Choi *et al.* describe algorithms for detecting block reference patterns of applications and applies different replacement policies to different applications depending on the detected reference pattern. The block reference patterns are classified as Sequential, Looping, Temporally clustered and Probabilistic.

We remark that while the works reported above in this Section studied the way data is read or written into memory, in this paper we are interested to know how the instructions are fetched in memory during execution. Data memory reference patterns are important for memory or computation performance reasons. For us, instruction memory reference patterns are important for the generation of synthetic mem-

ory reference traces.

3.1 Instruction Memory Reference Patterns

Memory reference patterns generated by instructions have been studied in the past by several researcher, for example by Abraham and Rau [1], who studied the profiling of load instructions using the Spec89 benchmarks. Their goal was to construct more effective instruction scheduling algorithms, and to improve compile-time cache management. Austin *et al.* [2] profiled load instructions while developing software support for their fast address calculation mechanism. They reported aggregate results from their experiments, not individual instruction profiles.

We recall that our approach for generating artificial traces of memory reference consists in the analysis of the real memory reference patterns generated by an application, and in building a stochastic model of the memory reference patterns. For this purpose the memory reference sequence must be described appropriately. Therefore we divide the original sequence in short frames, and we detect the type of the underlying execution. It is worth observing that the detection of loops, and the measure of the related period, highly depends on the frame size, because if the frame size is shorter than the period, it is impossible to detect that the address stream is periodic. However, in this case we still can capture the locality of the original memory reference sequence during the generation of synthetic memory references phase, as we will describe shortly.

Clearly, the first type of execution one can think about is *Sequential*. Thus we first use a sequentiality test, described shortly. If the frame is not sequential, we apply a periodicity test to see if the sequence is *Periodic*, which means that the instructions which generate the memory addresses is of type looping. For example, consider the following matrix multiplication code, which is of course made of nested loops.

```
// Multiplying matrices a (4x3) by b (3x4)
// storing result in 'result' matrix
  for(i=0; i<4; ++i)
     for(j=0; j<4; ++j)
        for(k=0; k<3; ++k)
           result[i][j]+=a[i][k]*b[k][j];
```

Figure 3. Matrix multiplication example

The instructions of this example make memory accesses that we capture with the PIN binary instrumentation framework [30]. To this purpose we use the *itrace* Pintool, that prints the address of every instruction that is executed. In Figure 4 we show a part of the memory reference pattern generated by the matrix multiplication code.

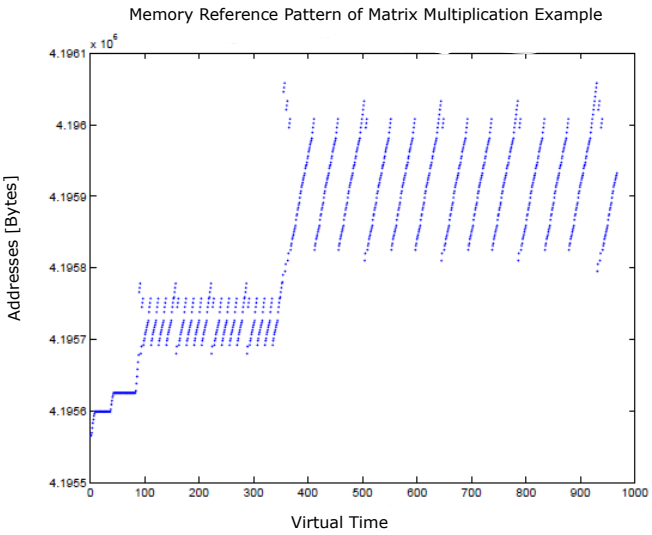

Figure 4. Memory references generated by the matrix multiplication example

We see a first burst of periodic addresses, from virtual time 100 to virtual time 350 approximately. This is the code which resets the $(4x4)$ *result* matrix. The actual matrix multiplication starts from virtual time 375 approximately.

The second example we discuss in this paper is related to indirect addressing used to access numeric vector. In the code included below, $c[]$ is a sparse array and $d[]$ is its index array. This example is taken from [16].

```
//access to sparse array
//c[]=sparse array. d[]=index array
  i=head;
  x=c[i];
  while(i){
       x += c[d[i]];
       i = d[i];
 }
```

Figure 5. Indirect address access example

In Figure 6 we show a part of the memory reference pattern generated by the numeric vector accessed with indirect addressing code. The pattern shows a periodic behavior, due to the *while* instruction. The access parts are only variable accesses.

Another aspect of this example we want to highlight is the following. We performed the generation of the index array in two ways, a deterministic and a random one. The deterministic generation code produces the memory references shown in Figure 6 while the memory references produced by random generation are shown in Figure 7.

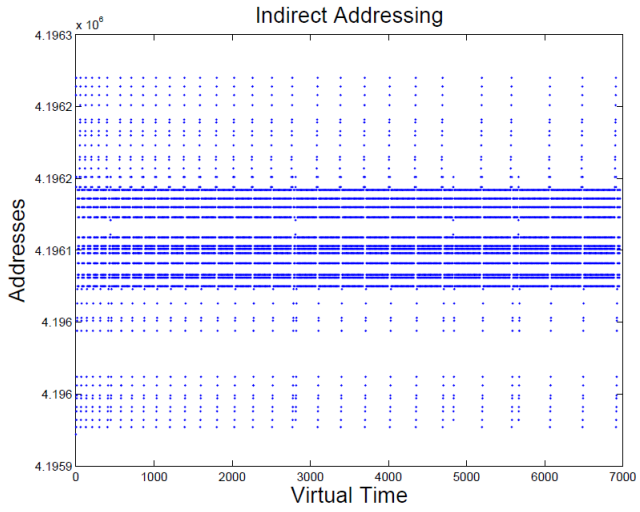

Figure 6. Memory references of the indirect access example via deterministic generation

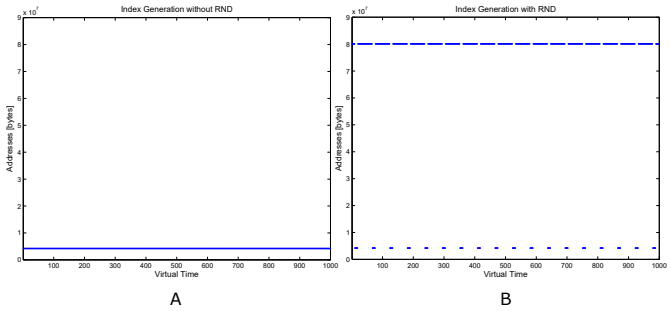

Figure 7. Memory references of the indirect access example via random generation

The difference among the two patterns is that in the random case we note that the addresses change abrutly at several virtual time instants. This is due to the calls to the subroutine *rnd*, which is a routine running at the user level. The amount of address change would have been much greater if a system call like an I/O routine had been made. Therefore, the third type of instruction is *Jump*. It is detected when the value of the addresses change greatly during a frame, which can be detected using a threshold. A *Jump* can be due to a branch in the code, or to a subroutine call or return.

The fourth type of instruction sequence is *Random*. It can be detected using a test for randomness. If no test gives a reliable output, then the frame is established to come from a sequence of instructions called *Other*, which is the fifth execution type.

3.2 Automatic Classification of Memory Reference Patterns

In this Section we describe the algorithms we used for the four tests.

1. *Sequential Pattern.* We classify the frame execution as *Sequential* as explained in the following. The values x_i of the memory reference addresses in a frame of length N are represented by the array $Frame$ reported in (1).

$$Frame = \left[x_i, x_{i+1}, x_{i+2}, \ldots, x_{i+N-1}, x_{i+N}\right] \quad (1)$$

The differences between adjacent memory address reference values are represented by the array Δ reported in (2).

$$\Delta = \left[(x_{i+1} - x_i), (x_{i+2} - x_{i+1}), \ldots, (x_{i+N} - x_{i+N-1})\right] \quad (2)$$

If all the values contained in the array Δ are positive, then $Frame$ is monotonic ascendent and it is classified as *Sequential*. Note that in this way a sequential frame can contain also ascending jumps. We assume that the monotonic test is performed by a software routine whose input argument is $Frame$. Our routine is called $Sequential(Frame)$, and has a boolean output, namely $true$ if the frame is sequential, $false$ otherwise. The $Slope$ value of sequential frames is easily found as the inclination angle of sequential patterns. $Slope$ sequences are then used to incrementally train the Hidden Markov Model $HmmS$ using a routine $HmmS = Inc\_Train(HmmS, Slope)$.

2. *Periodic Pattern.* The frame periodicity, and hence its *period*, is determined with standard spectral techniques used in signal processing for looking for signal harmonicity [28]. More precisely, given a frame of memory reference addresses as reported in (1), we weight its values with an *Hamming Window* [34], and we compute a Fast Fourier Transform [9] on it. The periodicity is detected by finding the relevant peaks in the spectrum amplitude and by looking for an harmonic structure of the peaks. For example, in Figure 8 we show a frame of size equal to 100 virtual ticks, taken from a memory reference sequence, with a clear periodic pattern. In the right pattern of Figure 8 we show the spectral amplitude of the windowed frame. In this case, the harmonicity can be easily detected. The first harmonic is the fundamental frequency of the frame is called f_0.

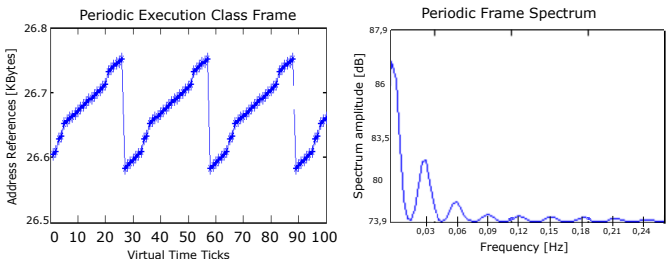

Figure 8. Spectral analysys of a periodic frame.

Since the harmonic structure shows a fundamental frequency equal to $f_0 = 0.03$ *Hz*, the period of the periodic pattern is evaluated as $1/f_0 = 33,3$. In our case the frame periodicity is computed by a software routine we call $Bounce(Frame)$. Also this routine has a Boolean output, namely $true$ if the frame is periodic and $false$ otherwise. The value of the period is estimated by the routine called $FindPeriod(Frame)$. The values of periods are used to incrementally train the Hidden Markov Model $HmmP$ as follows $HmmP = Inc\_Train(HmmP, Period)$.

3. *Random Pattern.* Many tests have been devised to verify the hypothesis of randomness of a series of observations, i.e. the hypothesis that N independent random variables have the same continuous distribution function [25].

   Our randomness test belongs the class of quick tests of the randomness of a time-series based on the sign test and variants [3]. This class of tests considers a series of N memory reference observations as that reported in (1) and the difference array reported in (2). If the observations are in random order, the expected number of plus or minus signs in (2) is $(N-1)/2$. The variance of (2) is $(N+1)/12$ and the distribution rapidly approaches normality as N increases. We then compute mean and variance of the sequence shown in (2) and we infer the frame randomness based on the similarity of the computed mean and variance with the expected ones. More precisely, we estimate the frame randomness with the routine called $Random(Frame)$.

   Once the randomness of a frame is established, its statistical distribution should be estimated for synthetic generation purposes. The discrete statistical distributions of the random variables $x_i$of the i-th frame are estimated by computing the Histograms of the frame itself. In a first step the original trace is analyzed until enough random frames are collected. For each random frame, its Histogram is computed. These Histograms divide the minimum – maximum range of the frame values in sixteen Bins, whose size is clearly $(max - min)/16$. Each Bin contains the number of values occurring in each interval divided by the frame size, say N, in order that the cumulative sum of Histogram is one. Complete information about the i-th random frame is contained in the $Stat(i)$ array reported in (3), which concatenates the Histogram values with the max and min values of the frame. In (3), $h_k(i)$ is the k-th Histogram value out of sixteen, i is the random frame index and $max(i), min(i)$ are the maximum and minimum of the i-th random frame.

$$Stat(i) = \big[h_1(i), h_2(i), \ldots, h_{16}(i), max(i), min(i)\big] \tag{3}$$

   All the obtained $Stat(i)$ arrays are combined in a $Codebook$ structure using standard clustering techniques [15]. In this way, all the random frames in the trace can be represented. We use a routine called $CodeBook = CB(H, max, min)$ for that purpose. Vector quantization of $Stat(i)$ means that each random frame is represented by an the index that corresponds to a minimum Euclidean distance between $Stat(i)$ in the trace and the $Codeword$ corresponding to the index. We code Random frames by the routine $Code = VQ(CodeBook, H, max, min)$. The code sequence is used to incrementally train the Hidden Markov Model $HmmR$ with the routine $HmmR = Inc\_Train(HmmR, Code)$.

4. *Jump Pattern.* The determination of $Jump$ patterns is straightforward. The difference between the values of the last and the beginning memory reference addresses of the frame is computed. If the difference is greater than a pre-established threshold the frame is classified as *Jump*. The Jump values are used to incrementally train the Hidden Markov Model $HmmJ$. We decide if the frame contains a jump or not with the routine $Bounce(Frame)$. The jump value is used to incrementally train the Hidden Markov Model $HmmJ$ with a routine $HmmJ = Inc\_Train(HmmJ, Code)$.

4 Experimental Results

We want to study if the algorithm is able to capture enough locality from the original traces. The simplest way to do that is to compare cache miss rate curves. We performed such experiments using a cache simulator, in particular the *Dinero IV* [22] and the benchmark suite SPEC2000 [20, 33]. Even if this benchmark suite has been officially discontinued by SPEC, still it is well suited to our purposes, as it is less demanding than more recent benchmarks, like SPEC2006. In Figure 9, Figure 10, Figure 11, and Figure

12, we report the miss-rate results for the `crafty`, `gzip`, `twolf`, `vortex` SPE2000 benchmarks.

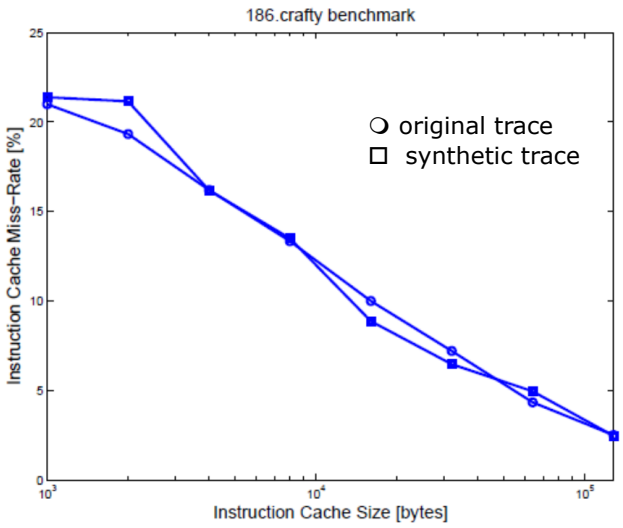

Figure 9. Original vs. synthetic instruction cache miss-rates for `crafty` benchmark

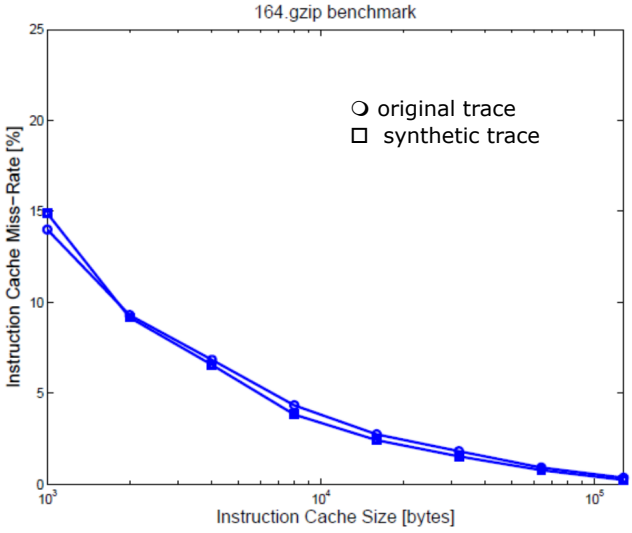

Figure 10. Original vs. synthetic instruction cache miss-rates for `gzip` benchmark

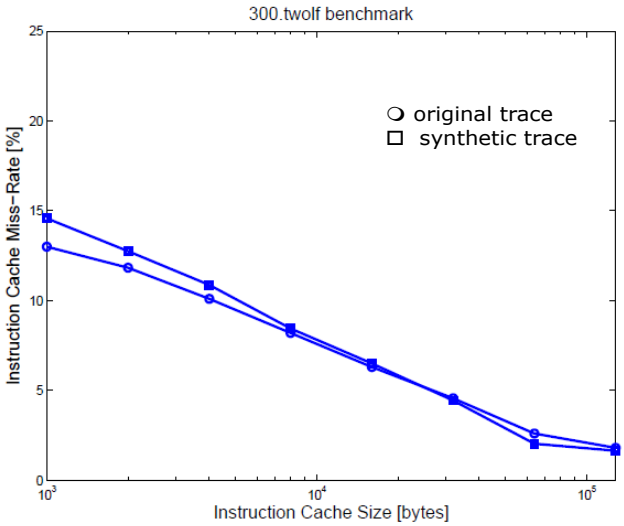

Figure 11. Original vs. synthetic instruction cache miss-rates for `twolf` benchmark

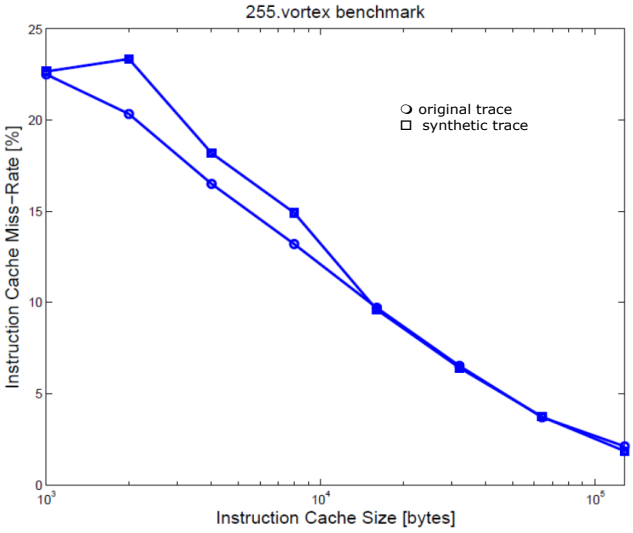

Figure 12. Original vs. synthetic instruction cache miss-rates for `vortex` benchmark

5 Final Remarks and Future Work

In this paper we describe an approach for workload characterization using ergodic hidden Markov models. The page references sequences produced by a running application are divided into short virtual time segments and used to train an HMM which models the sequence and is then used for run-

time classification of the application type and for synthetic traces generation. The main contribution of our approach are on one hand that a run-time classification of the running application type can be performed and on the other hand that the applications behavior are modeled in such a way that synthetic benchmarks can be generated.

As an extension, one can substitute *HnHMM* with stream classification methods, e.g. [29], or streaming sequential pattern mining approaches, e.g. [18], to allow for a batch-free adaptation to the sequences produced by programs during the execution, by also considering Cloud infrastructures (e.g., [11]) and big data performance issues (e.g., [12, 10]), for instance. A promising further direction that we want to additionally investigate is to improve the synthetic trace generation by considering the end-to-end context of the process that generates sequences out of program execution. In this case, application of online stream process mining will help in discovering the underlying process and adapt to it in real time [17], also following conventional approaches as regards the main adaptive model (e.g., [4]).

References

[1] S. G. Abraham and B. R. Rau. Predicting load latencies using cache profiles. *Internal Report HPL94110, Compiler and Architecture Research*, pages 1–44, 1996.

[2] T. M. Austin, D. N. Pnevmatikatos, and G. S. Sohi. Streamlining data cache access with fast address calculation. In *Proceedings of 22nd ISCA, Santa Margherita Ligure, Italy, June 22-24, 1995*, pages 369–380, 1995.

[3] C. Cammarota. The difference-sign runs length distribution in testing for serial independence. *Journal of Applied Statistics*, pages 1–11, 2010.

[4] M. Cannataro, A. Cuzzocrea, and A. Pugliese. A probabilistic approach to model adaptive hypermedia systems. In *Proceedings of the International Workshop for Web Dynamics*, pages 12–30, 2001.

[5] A. Chaurasia and V. K. Sehgal. Optimal buffer-size by synthetic self-similar traces for different traffics for noc. *SIGBED Review*, 12(3):6–12, 2015.

[6] J. Chen and R. M. Clapp. Astro: Auto-generation of synthetic traces using scaling pattern recognition for MPI workloads. *IEEE Trans. Parallel Distrib. Syst.*, 28(8):2159–2171, 2017.

[7] J. Choi, S. H. Noh, S. L. Min, E. Ha, and Y. Cho. Design, implementation, and performance evaluation of a detection-based adaptive block replacement scheme. *IEEE Trans. Computers*, 51(7):793–800, 2002.

[8] A. N. M. I. Choudhury, K. C. Potter, and S. G. Parker. Interactive visualization for memory reference traces. *Comput. Graph. Forum*, 27(3):815–822, 2008.

[9] W. Cochran, J. Cooley, D. Favin, H. Helms, R. Kaenel, W. Lang, G. Maling, D. Nelson, C. Rader, and P. Welch. What is the fast fourier transform? *Proceedings of the IEEE*, pages 1664 – 1674, 1967.

[10] A. Cuzzocrea. Accuracy control in compressed multidimensional data cubes for quality of answer-based OLAP tools. In *18th International Conference on Scientific and Statistical Database Management, SSDBM 2006, 3-5 July 2006, Vienna, Austria, Proceedings*, pages 301–310, 2006.

[11] A. Cuzzocrea, G. Fortino, and O. F. Rana. Managing data and processes in cloud-enabled large-scale sensor networks: State-of-the-art and future research directions. In *13th IEEE/ACM CCGrid, Delft, Netherlands, May 13-16, 2013*, pages 583–588, 2013.

[12] A. Cuzzocrea, F. Furfaro, and D. Saccà. Enabling OLAP in mobile environments via intelligent data cube compression techniques. *J. Intell. Inf. Syst.*, 33(2):95–143, 2009.

[13] A. Cuzzocrea, E. Mumolo, M. Hassani, and G. M. Grasso. Towards effective generation of synthetic memory references via markovian models. In *Proceedings of the 42nd IEEE Computer Society Signature Conference on Computers, Software and Applications, COMPSAC 2018, Tokyo, Japan, July 23-27, 2018*, 2018.

[14] D. Ferrari. On the foundations of artificial workload design. In *Proceedings of 1984 ACM SIGMETRICS, Cambridge, Massachusetts, USA, August 21-24, 1984*, pages 8–14, 1984.

[15] R. Gray. Vector quantization. *IEEE ASSP Magazine*, pages 4 – 29, 1984.

[16] L. Harrison. Examination of a memory access classification scheme for pointer-intensive and numeric programs. In *Proceedings of 10th ICS, Philadelphia, PA, USA, May 25-28, 1996*, pages 133–140, 1996.

[17] M. Hassani, S. Siccha, F. Richter, and T. Seidl. Efficient process discovery from event streams using sequential pattern mining. In *IEEE SSCI, Cape Town, South Africa, December 7-10, 2015*, pages 1366–1373, 2015.

[18] M. Hassani, D. Töws, A. Cuzzocrea, and T. Seidl. BFSP-Miner: an effective and efficient batch-free algorithm for mining sequential patterns over data streams. *International Journal of Data Science and Analytics*, Dec 2017.

[19] P. Heildelberger and H. Stone. Parallel trace-driven simulation by time partitioning. In *Proceedings of the Winter Simulation Conference*, pages 734–737, 1990.

[20] J. Henning. Spec cpu2000: measuring cpu performance in the new millennium. *Computer*, pages 1–44, 2000.

[21] M. A. Holliday. Techniques for cache and memory simulation using address reference traces. *Int. Journal in Computer Simulation*, 1(2), 1991.

[22] M. D. H. Jan Elder. Dinero iv trace-driven uniprocessor cache simulator, 2003.

[23] B. Jang, D. Schaa, P. Mistry, and D. R. Kaeli. Exploiting memory access patterns to improve memory performance in data-parallel architectures. *IEEE Trans. Parallel Distrib. Syst.*, 22(1):105–118, 2011.

[24] N. E. Jivan and M. R. Dagenais. A stateful approach to generate synthetic events from kernel traces. *Adv. Software Engineering*, 2012:140368:1–140368:12, 2012.

[25] D. E. Knuth. *The art of computer programming, Volume II: Seminumerical Algorithms*. Addison-Wesley, 1998.

[26] D. Lee, J. Choi, J. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. Kim. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Trans. Computers*, 50(12):1352–1361, 2001.

[27] J. Lee, C. Park, and S. Ha. Memory access pattern analysis and stream cache design for multimedia applications. In *Proceedings of 2003 ASP-DAC, Kitakyushu, Japan, January 21-24, 2003*, pages 22–27, 2003.

[28] T. Lobos, Z. Leonowicz, and J. Rezmer. Harmonics and interharmonics estimation using advanced signal processing methods. In *Harmonics and Quality of Power, 2000. Proceedings. Ninth International Conference on*, pages 335–340, 2000.

[29] Y. Lu, M. Hassani, and T. Seidl. Incremental temporal pattern mining using efficient batch-free stream clustering. In *Proceedings of 29th SSDBM, Chicago, IL, USA, June 27-29, 2017*, pages 7:1–7:12, 2017.

[30] C. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of 2005 ACM SIGPLAN PLDI, Chicago, IL, USA, June 12-15, 2005*.

[31] D. Nicol, A. Greenberg, and B. Lubachevsky. Massively parallel algorithms for trace-driven cache simulation. In *Proceedings of the 6th workshop on Parallel and Distributed Simulation (1992)*, pages 3–11, 1992.

[32] A. J. Niessen and H. A. G. Wijshoff. Address reference generation in a memory hierarchy simulator environment, 1995.

[33] C. Niki, J. Thornock, and K. Flanagan. Using the bach trace collection mechanism to characterize the spec2000 integer benchmarks. In *Proceedings of the Third IEEE Annual Workshop on Workload Characterization*, pages 369–380, 2000.

[34] P. Podder, T. Z. Khan, M. H. Khan, and M. M. Rahman. Comparative performance analysis of hamming,hanning and blackman window. *International Journal of Computer Applications (0975 8887)*, 96(18):1–7, 2014.

[35] H. S. Stone. *High-performance computer architecture (3. ed.)*. Addison-Wesley, 1993.

[36] D. Thiébaut, J. L. Wolf, and H. S. Stone. Synthetic traces for trace-driven simulation of cache memories. *IEEE Trans. Computers*, 41(4):388–410, 1992.