An Effective and Efficient Algorithm for Supporting the Generation of Synthetic Memory Reference Traces via Hierarchical Hidden/Non-Hidden Markov Models

***Abstract*—Trace-driven simulation is a popular technique useful in many applications, as for example analysis of memory hierarchies or internet subsystems, and to evaluate the performance of computer systems. Normally, traces should be gathered from really working systems. However, real traces require enormous memory space and time. An alternative is to generate Synthetic traces using suitable algorithms. In this paper we describe an algorithm for the synthetic generation of memory references which behave as those generated by given running programs. Our approach is based on a novel Machine Learning algorithm we called Hierarchical Hidden/non Hidden Markov Model (HHnHMM). Short chunks of memory references from a running program are classified as Sequential, Periodic, Random, Jump or Other. Such execution classes are used to train an HHnHMM for that program. Trained HHnHMM are used as stochastic generators of memory reference addresses. In this way we can generate in real time memory reference streams of any length, wich mimic the behaviour of given programs without the need to store anything. It is worth noting that our approach can be extended to other applications, for example network or data storage systems. In this paper we address only the generation of synthetic memory references generated by instruction fetches. Experimental results and a case study conclude this paper.**

Keywords:

Trace-driven simulation, ergodic HMM, memory references, execution classes, synthetic memory references, spectral analysis.

I. 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 [29]. 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 [50] and parallel simulation [27] [42] [3]. 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 an 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 [18] [51], have highlighted the difficulty to exactly describe original characteristics of the memory references, such as locality, with analytic models.

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 behaviour of the execution is learned by the model to ensure by machine learning that the artificial sequence will mimic the behaviour 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.

The rest of this paper is organized as follows. In Section II we report some previous research in synthetic trace generation area. In Section III our approach for the generation of synthetic traces is highlighted and in Section IV the reference patterns generated by instructions in execution are discussed. Section V deals with our Hierarchical Hidden/non Hidden Markov Models while Section VI we describe how the synthetic traces are generated. In Section VII some experimental results are reported and in Section VIII a case study is described. Finally, Section IX contains some final remarks.

II. Related Work

Noonnburg and Shen presented in [45] a framework which models the execution of a program on a particular architecture with a Markov chain. Moreover, Noonburg and Shen don't model memory dependencies and assume only one dependency per instruction. Carl and Smith [9] propose a hybrid approach, where a synthetic instruction trace was generated

based on execution statistics and fed into a trace-driven simulator. In [17], Eeckhout *et al.* present an algorithm which is a continuation of the work initiated by Carl and Smith by suggesting several improvements: incorporating memory dependencies, using more detailed statistical profiles and guaranteeing syntactical correctness of the synthetic traces. In particular they estimate instruction mixes, branches and dependencies to build a detailed profile. As a result, the performance prediction reported in this paper are far more accurate than those reported in [9].

One of the earliest authors that proposed the generation of memory references using stochastic approaches was Denning *et al.* [15]. They exploited the Independent Reference Model (IRM), proposed the previous years by the same authors, based on the assumption that memory references are independent random variables with stationary probabilities. However, IRM fails to capture the locality of memory references as it appears in a real trace. Thiebaut *et al.* [51] describe stochastic approaches for generating synthetic address traces that produce good replication of the locality of reference presented in real programs. They introduce locality in memory references by modeling the probability distribution of the jumps between consecutive memory references and by using such probability distribution to generate new memory references. Their algorithm uses the working-set size and locality of reference. Since the working-set changes during the execution of a program, this value is adaptively estimated.

Agarwal *et al.* describe in [30] an algorithm based on a two-state Markov chain. The first state produces sequential memory references and the second state generates random references. State transition is driven by data extracted from the real trace. However, the authors show that also their algorithm does not capture sufficient temporal locality.

Sorenson *et al.* [49] highlight the need to capture both spatial and temporal localities from a real trace. They analyze various known ways to visualize the locality in real traces and use them to evaluates some existing synthetic trace models. The authors study the Least Recently Used Stack Model (LRUSM) and other models but report poor cache performance.

Berg *et al.* [4] use the reuse distance to capture trace locality and applies the distribution to a probabilistic cache model to estimate the miss ratio of fully-associative caches. The reuse distance is the number of intervening memory references between identical references. However, the assumption that the larger the reuse distance, the higher the probability of a cache line removal is not necessarily true, at least for synthetic trace generation. The intermediate accesses could all be to the same memory location and a large reuse distance would not reflect this pattern.

Mattson *et al.* in [41] study the locality in terms of LRUSM, which is a natural representation of least recently used behaviour. LEUSM is based on the stack distance, which is the number of unique intervening memory references between identical references and is a very effective measure of temporal locality.

Grimsrud analyse in [21] the efficiency of the stack distance model in preserving temporal locality using his locality surfaces.

In [5] Brehob *et al.* use the stack distance model to implement a probabilistic cache model to evaluate miss ratio.

However, the works described in [41], [21], [5] do not analyse the efficacy of the LRUSM in the generation of synthetic traces for trace-driven simulation of cache memory. In this paper, we implement an algorithm adapted from the LRUSM and use it to characterise the spatio-temporal characteristics of application workloads. The profile data is passed to a Markov stochastic model which generates memory references through a dynamically ordered FIFO scheduler driven by a pseudo-random number generator.

III. 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.

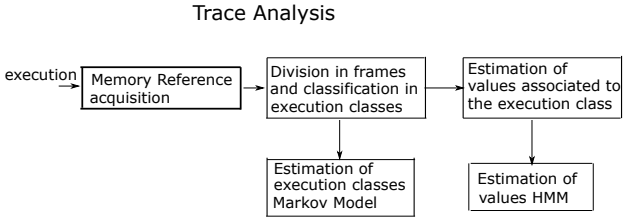

Fig. 1: Trace analysis algorithm

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 2.

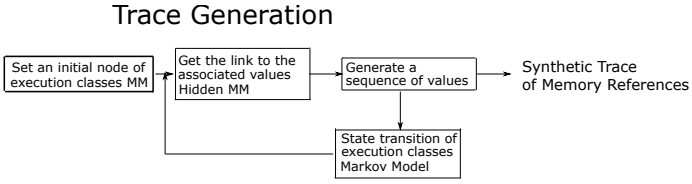

Fig. 2: Synthetic trace generation algorithm

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

IV. 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. in [6] Brewer *et al.* developed software tools to assist in formulating correct algorithms for high-performance computers and to aid as much as possible the process of translating an algorithm into an efficient implementation on a specific machine. Tools to understand memory access patterns of memory operations performed by running programs are described also by Choudhury *et al.* in [13]. Such studies are directed towards the optimization of data intensive programs such as those found in scientific computing.

Other works, for example [22], [32], [37] and [43], 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 [22] 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 Jongmoo Choi *et al.*, namely [10], [11], [33], [36], [12], analyze streams of disk block requests. Jongmoo 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 memory reference traces.

A. 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 consist 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 dependson 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 testy, described shortly. If the frame is not sequential, we apply a periocity 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];
```

Fig. 3: Matrix multiplication example

The instructions of this example make memory accesses that we capture with the PIN binary instrumentation framework [40]. 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.

We see a first burst of periodic addresses, from virtual time 100 to virtual time 350 approximately. This is the code which

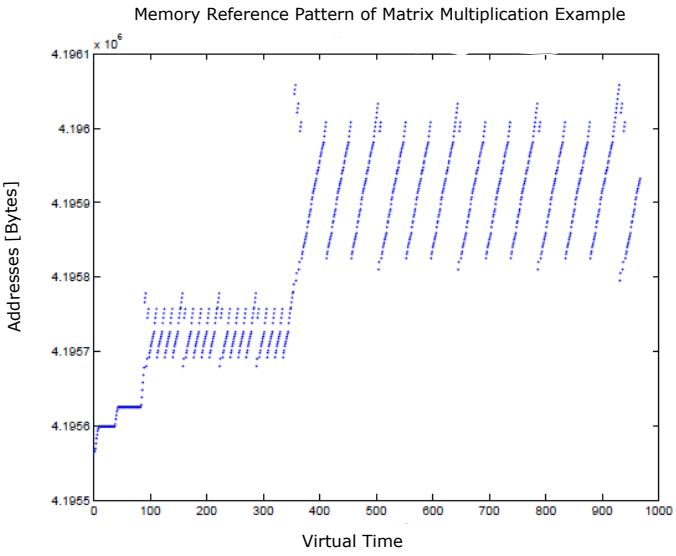

Fig. 4: Memory references generated by the matrix multiplication example

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 [22], page 8.

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

Fig. 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 behaviour, 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.

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

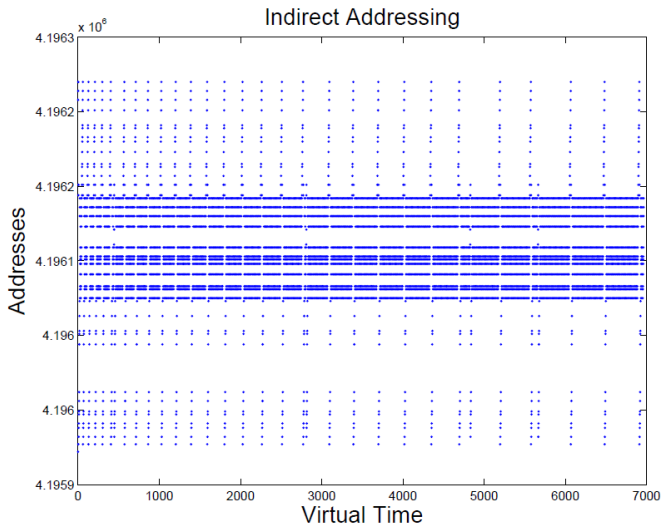

Fig. 6: Memory references of the indirect access example via deterministic generation

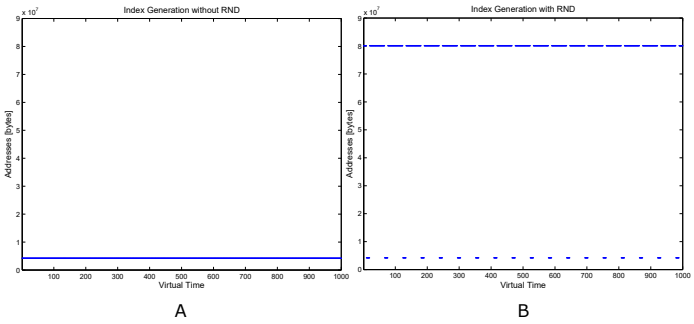

Fig. 7: Memory references of the indirect access example via random generation

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.

B. 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).

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

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 [38]. More precisely, given a frame of memory reference addresses as reported in (1), we weight its values with an *Hamming Window* [46], and we compute a Fast Fourier Transform [14] 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.

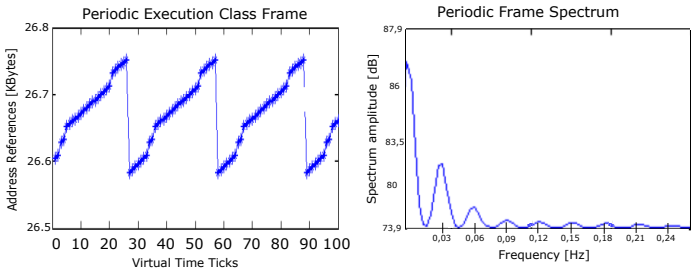

Fig. 8: Spectral analysys of a periodic frame.

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 [34].
Our randomness test belongs the class of quick tests of the randomness of a time-series based on the sign test and variants [8]. 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.

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

All the obtained $Stat(i)$ arrays are combined in a $Codebook$ structure using standard clustering techniques [20]. 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 $HmmJ = Inc\_Train(HmmJ, Code)$..

The algorithm for Automatic Classification of Memory Reference Patterns is described in the pseudo-code reported in Algorithm 1. In Algorithm 1 we assume that the memory reference stream is divided in short frames with a length ranging from 20 to 100 virtual time ticks typically. We find the right length empirically during an initial analysis of part of the original trace. The initial analysis is also needed for building the CODEBOOK. The symbols act_state, pre_state are the actual and previous states of the first level Markov Model with five states, namely S, P, R, J and O. The symbols $HmmS, HmmP, HmmR, HmmJ, HmmO$ are the five Hidden Markov Models which represent the $Slopes$ sequences for $Sequential$, the $Period$ sequences for $Periodic$, the Codeword sequences for $Random$ frames, $Jump$ sequences of $Bounced$ frames and $Other$ frames sequences of execution classes respectively.

V. Hidden/Non Hidden Markov Models (HnHMM)

A. Modeling Memory References by HnHMM

A memory reference sequence generated by an application is divided in frames and the execution in each frame is classified as Sequential, Periodic, Random, Jump or Other. Each classified execution is accompanied by a sequence of number. For example each sequential frame is accompanied by its slope, each periodic frame by its period, each random frame by its histogram, it Jump frame by the value of the jumps and if he frame is classified as Other, by the frame itself. The situation is represented for example in Figure 9. In this example, we have a sequence of three sequential frames whose slopes are $4, 5, 4$ followed by five periodic frames whose periods are $8, 8, 8, 9, 8$.

To model such complex signal we use a particular form of Markov Models, the Hierarchical Markov Models.

B. HnHMM Topology

Hierarchical Hidden/Non Hdden Markov Models (HHnHMM) are multi-level stochastic models devoted to generation of stochastic processes. Like Hierarchical Hidden Markov Models [35], [52], [7], [19], [48], HHnHMM emits sequences rather than a single symbol. As shown in Figure 10, HHnHMM are formed by a first level by a Markov Model, where the state sequence is observable, connected to several HMM where the states are not observable (Hidden) at a second level.

Algorithm 1 $FrameClassification$ Algorithm

Input: $Frame, pr\_state, N$;
Output: $HmmS, HmmP, HmmR, HmmJ$;
$HmmO, pre\_state$;
if $Sequential(Frame)$ **then**
  $act\_state = P$;
  $Slope = (Frame[N] - Frame[1])/N$;
  $HmmS = Inc\_Train(HmmS, Slope)$;
else if $Periodic(Frame)$ **then**
  $act\_state = S$;
  $Period = FindPeriod(Frame)$;
  $HmmP = Inc\_Train(HmmP, Period)$;
else if $Random(Frame)$ **then**
  $act\_state = R$;
  $H = Histogram(Frame)$;
  $CodeBook = CB(H, max, min)$;
  $Code = VQ(CodeBook, H, max, min)$;
  $HmmR = Inc\_Train(HmmR, Code)$;
else if $Bounce(Frame)$ **then**
  $act\_state = J$;
  $Jump = x_{i+N} - x_i$;
  $HmmJ = Inc\_Train(HmmJ, Jump)$;
else
  $act\_state = O$;
  $HmmO = Inc\_Train(HmmO, Frame)$;
end if
$MM = Inc\_Train(MM, act\_state, pre\_state)$;
$pre_s state = act_s state$;
return $HmmS, HmmP, HmmR, HmmJ$,
$HmmO, MM, pre\_state$;

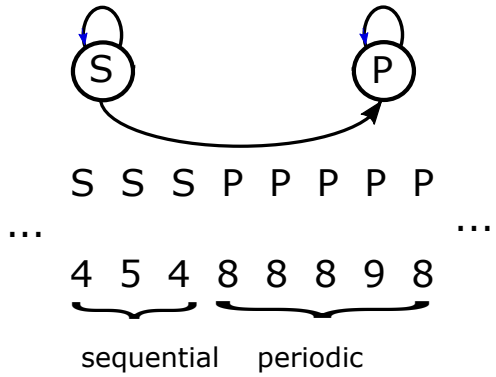

Fig. 9: Example of the memory reference sequence after frame division and classification

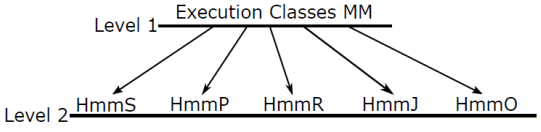

Fig. 10: The two levels of Hierarchical Hidden Markov Model

In Figure 11, finally, we report an extended topology of our HHnHMM. Each state of the MM points to a HMM. In this figure the hidden / observale structure is shown.

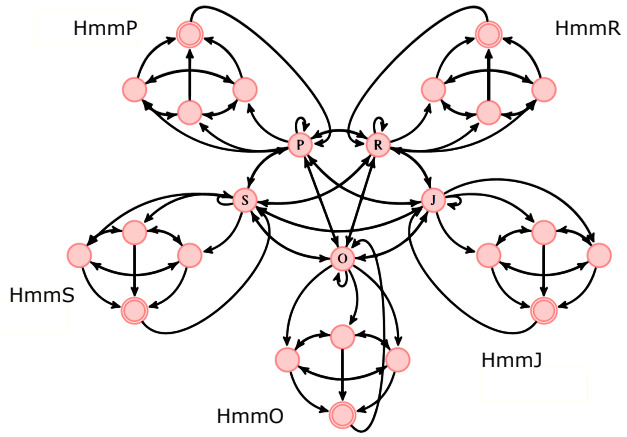

Fig. 11: Hierarchical Hidden Markov Model topology

C. Learning HHnHMM Parameters

First of all we give a formal description of our HHnHMM. Let Σ_1, Σ_2 be finite alphabets. In this paper we assume $\Sigma_1 = \{S, P, R, J, O\}$ and $\Sigma_2 \in N$. Let us denote Σ_2^* the set of all the possible finite strings over Σ_2. We call Observation sequence O, a finite string of the type $O = \gamma_1\gamma_2\cdots\gamma_T$ where the elements γ_i are pairs as shown in (4).

$$\gamma_i = (\alpha, \beta) \tag{4}$$

In the pairs shown in (4), $\alpha \in \Sigma_1$ and $\beta \in \Sigma_2^*$. For example, from Figure 9, $O = (S, 454)(P, 8898)$ and so forth. A state of HHnHMM is denoted by q_i^d where i is the state number and d is the level. In our case $d \in \{1, 2\}$ and $i \in \{1, 2, \cdots, max_{state}\}$ where max_{state} is the maximum number of states among the two levels. The first level MM is of course governed by a transition matrix $A^{q^d} = [a_{ij}^{q^d}]$ with $d = 1$. Each element of the matrix A^{q^d} is the transition probability to go from state i to state j, or $a_{ij}^q = Prob(q_j^d|q_i^d)$ with $d = 1$. At the second level there are some pretty standard HMM, each with its own transition matrix $A_k^{q^d} = a_{ij}^{q^d}$ with $d = 2$. In each element of $A_k^{q^d}$, with $d = 2$, the index k, $k = 1, 2, \cdots, |\Sigma_1|$, is the number of HMM, being $|\Sigma_1|$ the size of the set Σ_1. Moreover, q^d with $d = 2$, is the generic state of the second level HMM and $a_{ij}^{q^{d^k}} = Prob(q_j^{d^k}|q_i^{d^k})$ with $d = 2$, is the transition probability between states i and j in the k-th second level HMM.

The first level Markov Model has associated transition probabilities for going from the first to the second levels. As a matter of fact, these probabilities are the initial probability distribution of the second level HMM. In other words, there is an initial probability distribution matrix π^q whose elements are

$$\pi_{ij}^q = Prob\{q_j^2|q_i^1\} \tag{5}$$

where i is a state number of the first level MM and j is a state number of the second level HMM.

The training of the first level MM is performed in this way: in a matrix of size $|\Sigma_1| \times |\Sigma_1|$, we accumulate the number of transitions of each state to another state of the same level. In this way we measure the numbet of transitions of the state, say, 'S', to itself and to all the other states of the first level. The same holds for all the other states. All the matrix elements, at the end, are divided by the total number of transitions.

The second level Hmm are Ergodic, with a termination state, which is a state that is accessed when the input training Observation terminates. It is relatyed therefore to the length of the observation sequences. When the termination state is reached, then the second level HMM returns back to the first level state from which it started. The training of the second level HMM is performed using pretty standard approaches, which have the goal to estimate the HMM parameters in order to maximize $Prob(O|\lambda)$, where λ represents the HHnHMM parameters in short. $Prob(O|\lambda)$ is the likelihood that the model λ generates the observation sequence, as summarized hereafter.

The parameters are obtained through the EM approach; it is worth noting that the re-estimation formulas depend on the type of HMM, i.e. discrete or continuous. Let us denote by $\xi_t(i, j)$ the probability of being in state i at time t and in state j at time $t+1$, given the model and the observation sequence, and by $\gamma_t(i)$ the probability of being in state i at time t given the entire observation sequence and the model. Then, it can be shown [47] that for discrete HMM, the transition probabilities and the emission probabilities can be computed as reported in 6.

$$\begin{aligned} \pi_i &= \gamma_0(i) \\ a_{i,j} &= \frac{\sum_{t=1}^{T} \xi_{t-1}(i,j)}{\sum_{t=1}^{T} \gamma_{t-1}(i)} \\ b_i(k) &= \frac{\sum_{t \in \mathbf{O}_t = v_k} \gamma_t(i)}{\sum_{t=1}^{T} \gamma_t(i)} \end{aligned} \tag{6}$$

where v_k is the observed symbol occurring in state i, in our case the elements α described above.

In continuous HMMs, each state is not characterized by a discrete probability distribution but by a continuous probability density function, which is usually a multivariate Gaussian, i.e.:

$$f(\mathbf{x}) = \frac{1}{(2\pi)^{m/2}|\Gamma|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x}-\mu)^T\Gamma^{-1}(\mathbf{x}-\mu)\right) \tag{7}$$

where $\mathbf{x}$ is the M-dimensional aleatory vector, vector μ and matrix Γ respectively the mean and the variance of the M-dimensional Gaussian density.

It can be shown [47] that the re-estimation formula for mean and variance are of the form:

$$\mu_k = \frac{\sum_{t=1}^{T} \gamma_t(i,k)\mathbf{o}_t}{\sum_{t=1}^{T} \gamma_t(i,k)}$$
$$\gamma_{j,k} = \frac{\sum_{t=1}^{T} \gamma_t(i,k)(\mathbf{o}_t - \mu_{i,k})(\mathbf{o}_t - \mu_{i,k})^T}{\sum_{t=1}^{T} \gamma_t(i,k)} \quad (8)$$

VI. Synthetic Trace Generation

The approach described in this paper is divided in analysis/generation stages. In the analysis stage, original Memory Reference Addresses traces are divided in frames whose execution class is determined. Statistical models learn the classified sequence by training. In the synthesis stage, the trained statistical models are used to stochastic-ally generate the synthetic traces. In particular, we use a Markov Model to generate sequence of classes of execution captured from he original Memory Reference Addresses. The classes are $Sequential, Periodic, Random, Jump$ and $Other$. During trace generation, we start from a given initial Memory Address and from it we reconstruct the execution classes seen during analysis. For example, if we generate a $Sequential$ frame, we reconstruct a sequential sequence of addresses according to the $Slope$ value generated from $HmmS$. Similarly, to generate periodic frames we reconstruct a periodic sequence of address with a $Period$ equal to the value generated by $HmmP$. The general structure of the generation algorithm in reported in the Algorithm 2.

Let us discuss in more detail a couple of issues related to 2. First of all, we summarize in the following points how the generation of synthetic frames works.

1) Using the initial probability distribution for the first level MM, start with an initial state;
2) Upon entering in a first level MM state, we find the initial state of the associated second level HMM using the initial probability distribution reported in (5);
3) Generate a sequence β until the termination state T is reached;
4) Return to the first level MM and make a state transition;
5) Goto 2;

Secondly, let us summarize how the periodic frames are generated is periodic, its values are given by the *genPseq* routine The *genPseq* algorithm is described in Algorithm 3.

As reported in Algorithm 2, random sequences are generated by the $genRseq(start, index, N)$ routine. Its inputs are the $Codebook$ index, the starting address and the frame dimension. The Codewords are structured as reported in (3). Using that information, the Cumulative Distribution Function (CDF) is built from the Histogram and the max,min values by cumulative sum. The resulting CDF has values from 0 to 1 while the abscissa values range from min to max. Also min and max are stored in the Codeword. The CDF is stored in an array data structure where the array indexes are mapped from 0 to 1 and the corresponding array elements contain the values of the cumulative distribution function. It is well known

Algorithm 2 $SyntheticFrameGeneration$ Algorithm

Input: $start, secondLevelMMArray, currentState$;
Output: $syntheticMemoryReferenceFrame$;
$getActualState(firstLevelMM)$;
switch $(currentState)$
case S**:**
  $getNextSlope(HmmS)$;
  $seq = genSseq(start, slope, N)$;
  $start = last\_seq\_address$;
case P**:**
  $getNextPeriod(HmmP)$;
  $seq = genPseq(start, period, N)$;
  $start = last\_seq\_address$;
case R**:**
  $getNextCodeBookIndex(HmmR)$;
  $seq = genRseq(start, index, N)$;
  $start = last\_seq\_address$;
case J**:**
  $getNextJump(HmmY)$;
  $seq = genJseq(start, jump, N)$;
  $start = last\_seq\_address$;
case O**:**
  $getNextValues(HmmO)$;
  $seq = genOseq(start, values, N)$;
  $start = last\_seq\_address$;
end switch
$frame = seq$;
return $frame, start$;

Algorithm 3 $PeriodicFrameGeneration$ Algorithm

Input: $start, period, N$;
Output: $syntheticFrame, last\_address$;
$j = 0$;
for $i = 1toN$ **do**
  $m = MODULO(i, period)$;
  $j = j + 1$;
  if $(m == 0)$ **then**
    $syntheticFrame(i) = start$;
    $j = 1$;
  else
    $syntheticFrame(i) = start + j - 1$;
  end if
end for
$last\_address = syntheticFrame(i)$;
return $syntheticFrame, last\_address$;

that the inverse transformation method is a general method for generating random numbers from any probability distribution given its cumulative distribution function [16]. In practice, for each element of the frame, we take a random number from 0 and 1, we find the corresponding array index according to the mapping, and return the number contained in the array corresponding to the index. That is a random number generated according to the statistical distribution of the original frame.

The whole synthetic trace is built concatenating the frames generated as described in Section VI.

VII. 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* [31] and the benchmark suite SPEC2000 [28], [44]. 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 12, Figure 13, Figure 14, and Figure 15, we report the miss-rate results for the `crafty`, `gzip`, `twolf`, `vortex` SPE2000 benchmarks.

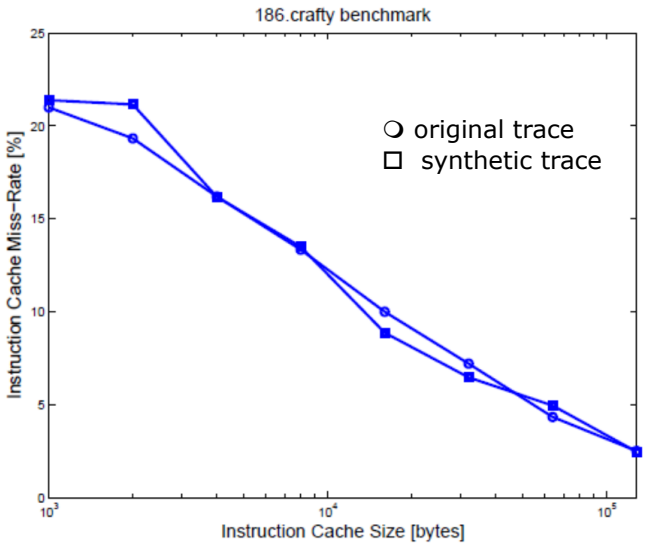

Fig. 12: Original vs. synthetic instruction cache miss-rates for `crafty` benchmark

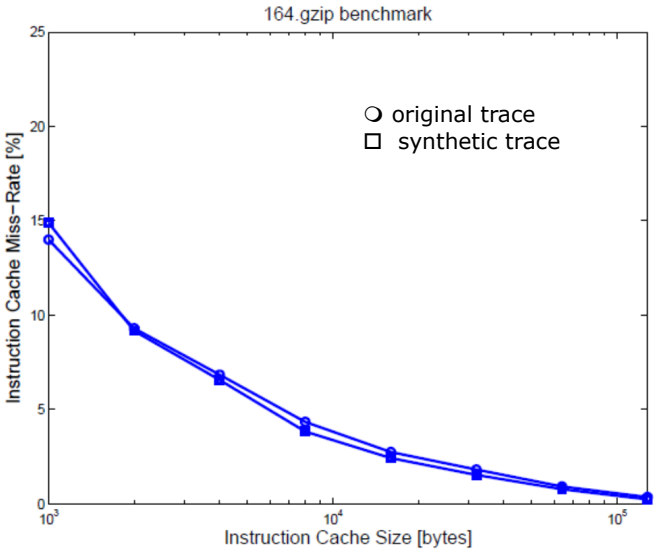

Fig. 13: Original vs. synthetic instruction cache miss-rates for `gzip` benchmark

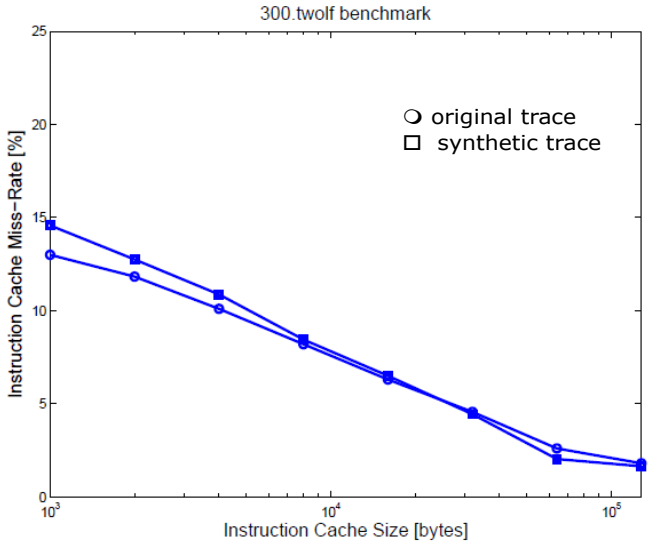

Fig. 14: Original vs. synthetic instruction cache miss-rates for `twolf` benchmark

VIII. Case Study

In this Section we present the analysis of trace `gcc` by the described algorithm.

In Figure 16, we report a section of the memory reference trace obtained from the `gcc` benchmark. The values of address references are reported in bytes versus time.

In Figure 17, we report the memory references obtained with the described algorithm. Clearly the trace is different from original. What we want to preserve, however, is the locality of the addresses.

In Figure 18, we can see an histogram of the execution classes extracted from the original sequence. The number of frames classified as *Other* is the majority, which means that in many case the automatic classifier is not able to decide if the frame is *sequentia*, *periodic*, *random* or *jump*. However, the algorithm classify half of the total frames.

In Figure 19, we report the cache miss rate of the synthetic trace compared with the original. This result shows that the algorithm is able to capture a good level of locality of `gcc`.

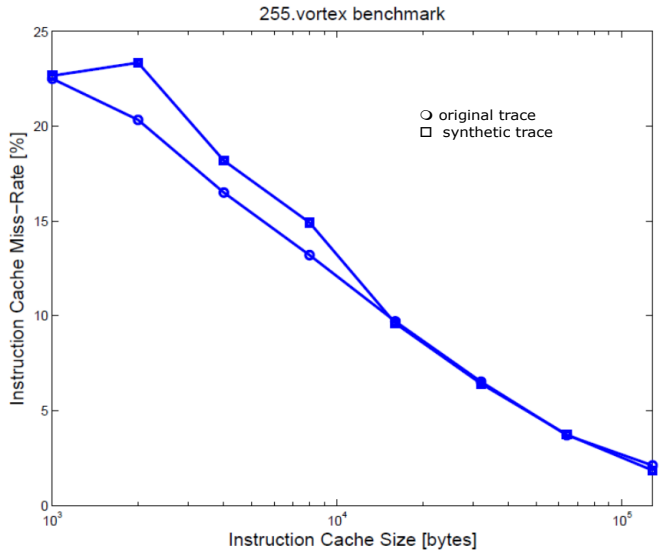

Fig. 15: Original vs. synthetic instruction cache miss-rates for `vortex` benchmark

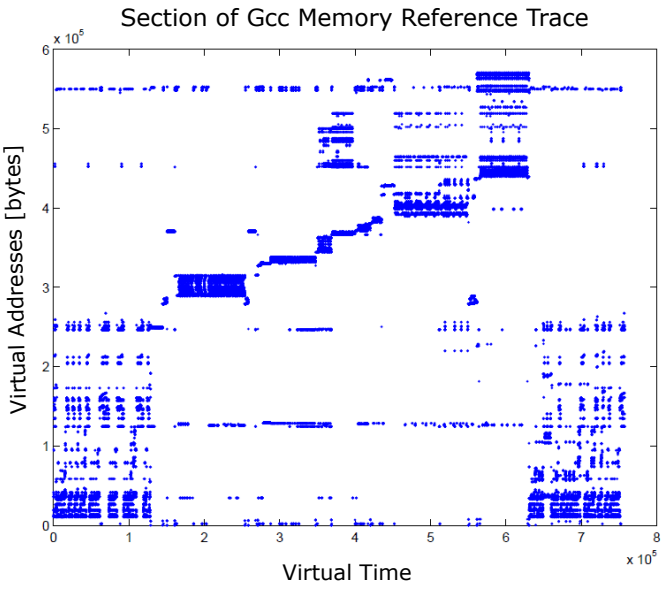

Fig. 16: `gcc` benchmark analysis: address references versus time

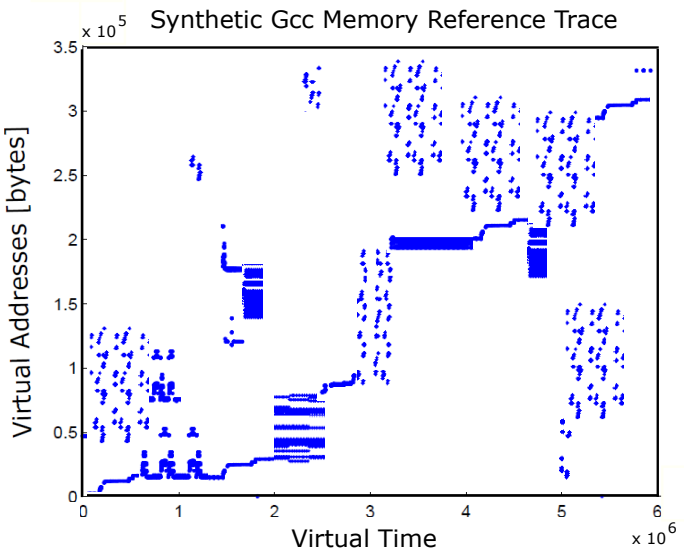

Fig. 17: `gcc` benchmark analysis: synthetic address reference

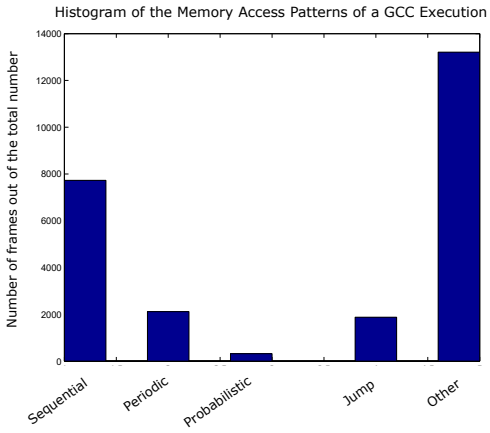

Fig. 18: `gcc` benchmark analysis: distribution of the execution classes

IX. 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. Using trace-driven simulation with SPEC2000 benchmarks, the mean classification rate is about 82% for each traces and about 76% using a single HMM to model a single application type. Many future developments of our approach are possible since what we propose in this paper – to use time-varying non-linear processing techniques to treat sequences produced by programs during execution – is a novel approach in computer architecture studies.

For instance, one can substitute *HnHMM* with stream classification methods e.g. [23], [26], [39] or streaming sequential pattern mining approach [25] to allow for a batch-free adaptation to the sequences produced by programs during the execution. 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

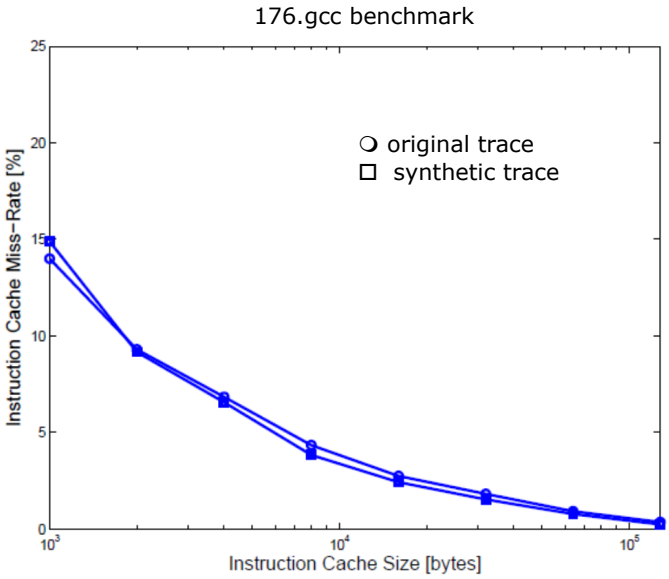

Fig. 19: `gcc` benchmark analysis: original vs. synthetic instruction cache miss-rates for `gcc`

case, application of online stream process mining will help in discovering the underlying process and adapt to it in real time [24].

REFERENCES

[1] S. G. Abraham and B. R. Rau. Predictingf low 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 the 22nd Annual International Symposium on Computer Architecture, ISCA '95, Santa Margherita Ligure, Italy, June 22-24, 1995*, pages 369–380, 1995.

[3] L. Barriga and R. Ayani. Parallel cache simulation on multiprocessor workstations. In *Proceeding of the 22nd International Conference on Parallel Processing*, pages 3–11, 1993.

[4] E. Berg and E. Hagersten. Statcache: a probabilistic approach to efficient and accurate data locality analysis. In *2004 IEEE International Symposium on Performance Analysis of Systems and Software, March 10-12, 2004, Austin, Texas, USA, Proceedings*, pages 20–27, 2004.

[5] M. Brehob and R. Enbody. An analytical model of locality and caching. *Michigan State University*, pages 1–9, 1999.

[6] O. Brewer, J. J. Dongarra, and D. C. Sorensen. Tools to aid in the analysis of memory access patterns for FORTRAN programs. *Parallel Computing*, 9(1):25–35, 1988.

[7] H. H. Bui, D. Q. Phung, and S. Venkatesh. Learning hierarchical hidden markov models with general state hierarchy. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California, USA*, pages 324–329, 2004.

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

[9] R. Carl and J. E. Smith. Modeling superscalar processors via statistical simulation. In *In Workshop on performance analysis and its impact on design*, pages 3–11, 1998.

[10] J. Choi, S. H. Noh, S. L. Min, and Y. Cho. An implementation study of a detection-based adaptive block replacement scheme. In *Proceedings of the 1999 USENIX Annual Technical Conference, June 6-11, 1999, Monterey, California, USA*, pages 239–252, 1999.

[11] J. Choi, S. H. Noh, S. L. Min, and Y. Cho. Towards application/file-level characterization of block references: a case for fine-grained buffer management. In *Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, Santa Clara, CA, USA, June 18-21, 2000*, pages 286–295, 2000.

[12] 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.

[13] 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.

[14] 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.

[15] P. J. Denning and S. C. Schwartz. Properties of the working set model. *Commun. ACM*, 15(3):191–198, 1972.

[16] L. Devroye. *Non- Uniform Random Variate Generation*. Springer Verlag, 1986.

[17] L. Eeckhout, K. D. Bosschere, and H. Neefs. Performance analysis through synthetic trace generation. In *International Symposium on Performance Analysis of Systems and Software*, pages 3–11, 2000.

[18] D. Ferrari. On the foundations of artificial workload design. In *Proceedings of the 1984 ACM SIGMETRICS conference on Measurement and modeling of computer systems, Cambridge, Massachusetts, USA, August 21-24, 1984*, pages 8–14, 1984.

[19] S. Fine, Y. Singer, and N. Tishby. The hierarchical hidden markov model: Analysis and applications. *Machine Learning*, 32(1):41–62, 1998.

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

[21] K. S. Grimsrud, J. K. Archibald, R. L. Frost, and B. E. Nelson. On the accuracy of memory reference models. In *Computer Performance Evaluation, Modeling Techniques and Tools, 7th International Conference, Vienna, Austria, May 3-6, 1994, Proceedings*, pages 369–388, 1994.

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

[23] M. Hassani, P. Kranen, R. Saini, and T. Seidl. Subspace anytime stream clustering. In *Conference on Scientific and Statistical Database Management, SSDBM '14, Aalborg, Denmark, June 30 - July 02, 2014*, pages 37:1–37:4, 2014.

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

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

[26] M. Hassani, D. Töws, and T. Seidl. Understanding the bigger picture: batch-free exploration of streaming sequential patterns with accurate prediction. In *Proceedings of the Symposium on Applied Computing, SAC 2017, Marrakech, Morocco, April 3-7, 2017*, pages 866–869, 2017.

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

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

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

[30] A. Hossain and D. J. Pease. An analytical model for trace cache instruction fetch performance. In *19th International Conference on Computer Design (ICCD 2001), VLSI in Computers and Processors, 23-26 September 2001, Austin, TX, USA, Proceedings*, pages 477–480, 2001.

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

[32] 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.

[33] J. M. Kim, J. Choi, J. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. Kim. A low-overhead, high-performance unified buffer management scheme that exploits sequential and looping references. In *4th Symposium on Operating System Design and Implementation (OSDI 2000), San Diego, California, USA, October 23-25, 2000*, pages 119–134, 2000.

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

[35] D. Kulic and Y. Nakamura. Incremental learning of human behaviors using hierarchical hidden markov models. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems, October 18-22, 2010, Taipei, Taiwan*, pages 4649–4655, 2010.

[36] 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.

[37] J. Lee, C. Park, and S. Ha. Memory access pattern analysis and stream cache design for multimedia applications. In *Proceedings of the 2003 Asia and South Pacific Design Automation Conference, ASP-DAC '03, Kitakyushu, Japan, January 21-24, 2003*, pages 22–27, 2003.

[38] 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.

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

[40] 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 the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*.

[41] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[42] 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.

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

[44] 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.

[45] D. B. Noonburg and J. P. Shen. A framework for statistical modeling of superscalar processor performance. In *Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture (HPCA '97), San Antonio, Texas, USA, February 1-5, 1997*, pages 298–309, 1997.

[46] 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.

[47] L. Rabiner and B. Juang. *Foundamentals of Speech Recognition*. Prentice Hall Signal Processing Series, 1993.

[48] C. A. Ronao and S. Cho. Recognizing human activities from smartphone sensors using hierarchical continuous hidden markov models. *IJDSN*, 13(1), 2017.

[49] E. Sorenson and J. K. Flanagan. Evaluating synthetic trace models using locality surfaces. In *International Workshop on Workload Characterization*, pages 3–11, 1992.

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

[51] 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.

[52] K. Wakabayashi and T. Miura. Topology estimation of hierarchical hidden markov models for language models. In *Natural Language Processing and Information Systems, 15th International Conference on Applications of Natural Language to Information Systems, NLDB 2010, Cardiff, UK, June 23-25, 2010. Proceedings*, pages 129–139, 2010.