Choosing Representative Slices of Program Execution for Microarchitecture Simulations: A Preliminary Application to the Data Stream

Authors:	Thierry Lafage (contact author)	André Seznec
Phone:	+33 2 99 84 25 01	+33 2 99 84 73 36
Fax:	+33 2 99 84 25 28	+33 2 99 84 25 28
E-mail:	Thierry.Lafage@irisa.fr	Andre.Seznec@irisa.fr

Address:

IRISA
Campus de Beaulieu
35042 RENNES CEDEX
France

Choosing Representative Slices of Program Execution for Microarchitecture Simulations: A Preliminary Application to the Data Stream

Thierry Lafage, André Seznec

6th July 2000

Abstract

Microarchitecture simulations are aimed at providing results representative of the behavior of a processor running an application. Due to CPU time constaints, only a few execution slices of a large application can be simulated. The aim of this paper is to propose a technique to choose a few program execution slices representative of the entire execution. We characterize the behavior of each consecutive slice executed. Then we use a statistical classification method to discriminate the execution slices and select the representative ones. In this paper, we detail this approach and apply it to the data stream. Using data cache simulations on the SPEC95 programs, we show that slices representing 1.46 % (average upon all the SPEC95 but one) of the overall program activity are as representative as trace sampling using a 10 % sampling ratio.

keywords: micro-architecture simulation, trace-driven simulation, on-the-fly simulation, trace sampling, clustering, classification, data stream characterization, data cache simulation.

1 Introduction

As the complexity of processors is always increasing, microarchitecture simulations using realistic applications consumes always more CPU time. Execution slowdowns to simulate out-of-order execution microprocessors are in the 1,000–10,000 range [3]. However, when exploring new architectural trends, microprocessor architects need their simulations to complete in a affordable amount of time. Therefore, either the target programs are run with reduced input data sets, or the simulations are executed over smaller portions of execution [21, 22, 23]. When using reduced input data sets, the simulation results may not accurately reflect a realistic activity of the processor, so we do not take it into consideration in the remaining of this paper.

To reduce simulation time on a realistic application and input data set, the simulation is often run over an arbitrary fixed number (e.g. a few billions) consecutive executed instructions (a *big slice*). In order to skip the initialization stage, simulation is often disabled during the execution of the first instructions (hundreds of millions, or billions) [4, 27].

Another way to reduce simulation time is the *trace sampling* technique as suggested by [18, 28] for cache simulations. Trace sampling consists in running the simulation over pseudo randomly chosen fixed-size slices of program execution, commonly referred to as *samples* (e.g. samples of 500000 references every 5 million references) or *clusters* [1].

In both cases, the amount of simulated data is determined by the execution time of the simulation, and/or by previous empirical results (for instance, a 10 % sampling ratio "is known" to give quite good results). Also, for both methods, the representativity of the simulated slices is very questionable.

Commonly used benchmarks have life times of several years (for instance, 3 years for the SPEC92 benchmarks, 5 years for the SPEC95 benchmarks). Microprocessor design projects are also very long efforts spanning over 5 or more years. Then it appears worthwhile spending time, once and for all, to select representative execution slices over which numerous time consuming simulations will be run.

In this paper we propose to use a classification method on several measures gathered over each consecutively executed program slice, in order to select the most representative ones for microarchitecture simulations.

The target program execution is first cut into fixed-size slices (e.g. one million instructions). Then metrics independent from the simulated microarchitecture implementation are applied to the program, and measures are reported for each execution slice.

This gives us a characterization of the dynamic program behavior which can be represented by multivariate statistical data (each *individual* is a slice of the execution). Then, we apply a classification method to group execution slices. For each class, we pick out the slice which is the most representative of the class (i.e. the closest to the class center). Finally, the selected slices are weighted by the representativeness of their class among all classes.

Our technique takes part of precise dynamic information gathered on the target programs for selecting representative slices. It also allows us to evaluate *a priori* the representativeness of the selected slices, through the computation of an *indicator*.

In order to validate our approach, we characterized the data stream of the SPEC95 benchmarks to run data cache simulations. However, a characterization of the instruction stream and data dependencies may be added as input to the classification tool to enable complete microprocessor simulations.

In the next section, we discuss related work on program slice selection for

[1] In the remaining of this paper we only use the term *samples*.

microarchitecture simulations. Section 3 develops our approach in detail. Section 4 is an application of the proposed method to the data stream of the SPEC95 benchmarks. Section 5 summarizes this study and presents directions for future development.

2 Related Work

Using reduced traces (or simulating on-the-fly partial program activity) is needed to perform simulations in an affordable amount of time. A great amount of work has been proposed to shorten overall simulation times.

First, trace sampling has been suggested to improve cache [18] or microprocessor [5] simulation speeds. This technique is known in statistics as *cluster sampling* [16]. Several fixed-size trace slices are picked out at regular or (pseudo-)randomly sized intervals and the simulations are run over them. The chosen slices may be more or less representative of the behavior of the target programs, depending on their size and number. Also, simulations results are biased because the state of the simulator at the beginning of a slice is different from the real state produced by full trace simulation. This effect is known as the *cold start effect*.

Similarly, our approach provides a mean to choose execution slices on which to perform simulations. These may also suffer cold start effects but the methods used in trace sampling to reduce it may also apply. In other respect, our approach takes advantage of dynamic program information to derive a representative slice set. In addition, we provide an indicator which gives an idea of the representativity of the selected slice set. This indicator does not need the simulations to be run.

A recent work proposes to use a small (i.e. 50 M. instructions) "representative" execution slice [25] for micro-architecture simulations. To this end, "*interval*" branch-misprediction data- and instruction-cache miss rates have been measured: i.e. the measures have been computed separately over each million-instruction interval in the target programs. The computed measures clearly exhibit initialization phases of programs. Then, for each program, a 50M. instruction execution slice is taken after the characterized initialization phase to further run time-consuming simulations. To validate the "representativeness" of the selected slices, each 50 M-instruction window has been compared to a 250 M-instruction simulation window running a cycle-accurate simulator of an out-of-order execution processor close to an Alpha 21264 [14].

We suspect such a method to be quite risky since the instruction slices are chosen manually and the slice size is arbitrary. Also the validation of the selected slices relies on longer slices which may not be representative of the whole behavior of the target programs (each program considered executed billions of instructions). At the end of subsection 4.4 we compare results from our approach and from a technique close to this, which consists in choosing an instruction slice which size represents 10 % of all the instructions executed. This technique, compared to ours and the trace sampling technique, lead to the more erroneous results.

The representativeness of the reduced trace (or the selected execution slices) when compared to the full original trace (or the complete program execution) is difficult to evaluate *a priori* and necessitates to characterize the behavior of the target program.

Profile-driven sampling [8] was proposed to obtain a representative reduced size trace. This technique uses a fine-grained program profile (frequency of execution of basic blocks) to filter the full trace. In the reduced trace, only a reduced number of occurrences of each executed basic block is kept. This way, statistics such as instruction mix distribution, and basic block size distribution would match those of the full trace. *Profile-driven sampling* takes advantage of dynamic program information to derive a reduced trace. However, the trace size reduction is specified as an input parameter of the method: it does not depend on the program characteristics but on the time available for simulation. Instead, the method we propose provides an indicator (see Section 3) which helps in choosing an adequate trace size reduction.

To evaluate the representativeness of a reduced trace against the full trace, Iyengar et al. [13] introduced the *R-metric*. The representativeness of a reduced trace (i.e. the result of the *R-metric* applied to a reduced trace) is an overall score. Based on the contents of the *R-metric*, a heuristic helps in generating reduced trace which are expected to have good *R-metric* scores.

The *R-metric* is based on some key characteristics of modern superscalar processors but it does not depend on the microarchitecture implementation details, as our metrics for data memory accesses do (see 4.1). On the other hand, the *R-metric* only evaluates a reduced trace *a posteriori*, while our approach uses the metrics to reduce the simulated program activity.

Time varying behavior of the SPEC95 benchmarks are presented in [24]. [24] reports simulation results for these programs each 100 million executed instructions for various modern architectural features: instructions per cycle (IPC), RUU occupancy, cache miss rate, branch prediction miss rate, address prediction miss rate and value prediction miss rate. This study points out that programs have widely different behaviors over time. For this reason, the sections of the program execution which are simulated must be accurately chosen to be representative of the program behavior as a whole. Our method is driven by this key idea, but we feel that program behavior must be characterized by metrics independent from the architectural implementation which is to be simulated. Moreover, our method actually select program parts to be simulated and evaluates their representativeness.

3 Representative Execution Slice Selection

We propose to use a characterization of the dynamic behavior of the target programs in order to select representative execution slices for microarchitecture simulations. This section presents in detail the process of selecting the slices, and an evaluation of their representativeness.

Characterization of the Dynamic Program Behavior. In order to characterize the dynamic behavior, we first divide the program execution into fixed-size slices (e.g. one million instructions). Then metrics are gathered on the program, and reported for each execution slice.

Choosing metrics to characterize this behavior is a difficult issue. For instance, for data cache behavior, one would like to capture the behavior of a large family of caches with distinct parameters such as associativity, size, line size, replacement policy, etc. For the CPU core behavior, one would like to be able to capture the behavior of different numbers of ALUs... We believe that metrics highly related to these behaviors, but independent from the implementation details can be found (e.g. see Section 4 for metrics related to the data stream).

Actual Slices Selection. Following commonly used statistics naming, each slice will be called an *observation* or *individual*, and each metric is considered as a set of *variables*. For an observation, each variable is a coordinate in a multidimensional space [2]. This representation of the characterization of the program behavior is a multivariate statistical data [10].

We apply to this multivariate statistical data a classification method to group points which have behaviors close to each others, with respect to the collected metrics. In each group, we retain the most *representative* slice: it is the point closest to the group center.

Simulation results on each chosen slice will be weighted by the representativeness of its class among all classes:

$$\text{Weight(slice)} = \text{Rep(class)} = \frac{\text{number of points in the class of this slice}}{\text{total number of points}} \qquad (1)$$

Representativeness of the Selected Slices. For a given classification, in a class, we define the representativeness of the selected point as the (Euclidean) distance between the selected point and the center of the class. Accordingly, we introduce for the whole "population", an indicator of the representativeness of all selected points: it is the weighted mean of the representativeness of the selected points in each class (the weights are those of Equation 1).

This indicator, called *wmdc* (weighted mean distances from centers), globally evaluates the representativeness of the selected slices among all execution slices.

[2] For this reason, in the remaining of this paper, we shall also call an observation a (measure) *point*.

4 An Application to the Data Stream

In this section, we present an application of the method described in the previous section to choose representative slices from a characterization of the data stream. We then measure the representativeness of the selected slices by comparing the results of data cache simulations run on all the program slices and run on the selected slices only.

We first detail architecture independent metrics to evaluate the data memory reference stream. Then, we present the classification tool we used for this experiment. Finally, experimental results on the SPEC95 benchmarks are reported and our method is compared to 1) simulations run over an arbitrary big slice, and 2) the systematic statistical trace sampling approach for data cache simulations.

4.1 Metrics Used

Cache memories [26] were introduced to take advantage of the spatial and temporal locality of memory references [12]. The metrics defined below characterize the temporal and spatial locality of data memory accesses for various line sizes in order to simulate a large range of cache configurations.

We first detail the metrics, then in a detailed example we emphasize the intuition behind them.

Measuring Temporal Locality. We evaluate temporal locality by counting the number of executed instructions between two accesses at the same address for each address in the program. This number is the least recently used distance expressed in terms of instructions executed (**LRUDI**) between two accesses at the same address.

Measuring Spatial Locality. Spatial locality is exploited in caches by the size of the line. In order to catch spatial locality information, we measure the temporal locality we defined above with *several* line sizes [3]. Spatial locality information is then characterized by the *difference* between the temporal locality distributions for several line sizes. Let us illustrate this on the following example.

Example. Let a reference stream be as follows (a is an address, numbers are in bytes): a (#1), $a+4$ (#2), $a+8$ (#3), $a+12$ (#4), $a+16$ (#5), $a+20$ (#6), $a+24$ (#7), a (#8), $a+4$ (#9), $a+8$ (#10), $a+12$ (#11), $a+16$ (#12), $a+20$ (#13), $a+24$ (#14). We assume that these references are done inside loops, so that the number of instruction executed between two references is fixed and equals 10. Note that this reference stream exhibits a good spatial locality.

- If the line size is 4 bytes, then each reference is done at a different line address. All 7 (twice referenced) addresses (i.e. references #8, #9, #10,

[3] For instance, the data at absolute address 'a' will be at line address: a $\gg \log_2$(line size).

#11, #12, #13, #14) have a LRUDI_4 of $10 \times 7 = 70$ instructions. For instance, the line address '$a + 16 \gg \log_2(4)$' is referenced for the second time (#12) 70 instructions after the first time (#5).

- If the line size is 16 bytes, then we assume references to $a, a + 4, a + 8, a + 12$ to match the same line address. Consequently, references to $a+16, a+20, a+24$ match the same line address. Reference #8 to a has an LRUDI_{16} of $(8-5) \times 10 = 30$ instructions; references #9, #10, #11 to $a+4, a+8$, and $a+12$ respectively have an LRUDI_{16} of 10 instructions because they match the same line address as a. Reference #12 to $a + 16$ (line address different from the 4 previous references) has an LRUDI_{16} of $(12-7) \times 10 = 50$ instructions...

The distributions of LRUDI_4 and LRUDI_{16} are very different. This difference represents the spatial locality: a 16 byte line takes advantage of the spatial locality, whereas a 4 byte line does not.

If, in contrast, each reference in the reference stream had been really far from the previous ones (poor spatial locality), then the distribution of LRUDI_{16} would have been the same as the distribution of LRUDI_4 (each reference would have matched a different line in both cases).

Data Representation. The multivariate statistical representation of a measured program is composed of individuals (the execution slices) and variables (measures). Given a line size *ls*, for each slice, and for each integer n, we gather the number of memory accesses which LRUDI_{ls} is between 2^n and $2^{n+1} - 1$[4]. For example, the LRUID_{16} is represented as follows:

$$
\text{Individuals}\left\{
\overbrace{
\begin{array}{c|ccccc|c}
\cdots & & & \mathrm{LRUDI}_{16} & & & \cdots \\
\hline
 & r^{16}_{1,1} & \cdots & r^{16}_{1,j_{16}} & \cdots & r^{16}_{1,n_{16}} & \\
 & \vdots & & \vdots & & \vdots & \\
\cdots & r^{16}_{i,1} & \cdots & r^{16}_{i,j_{16}} & \cdots & r^{16}_{i,n_{16}} & \cdots \\
 & \vdots & & \vdots & & \vdots & \\
 & r^{16}_{N,1} & \cdots & r^{16}_{N,j_{16}} & \cdots & r^{16}_{N,n_{16}} &
\end{array}
}^{\text{Variables}}
\right.
$$

The value of the j_{16}th variable for the ith observation ($r^{16}_{i,j_{16}}$) is the reference rate (among all references done during the ith execution slice) which LRUDI_{16} value is between $2^{j_{16}}$ and $2^{j_{16}+1} - 1$ instructions:

$$
r^{16}_{i,j_{16}} = \frac{\mathrm{Card}\left(\left\{x \in X_i / \mathrm{LRUDI}_{16}(x) \in [2^{j_{16}}; 2^{j_{16}+1} - 1]\right\}\right)}{\mathrm{Card}(X_i)}
$$

With: $X_i = \{\text{References referenced in the } i\text{th slice}\}$

Note that n_{16} is defined by: $\exists i, r^{16}_{i,n_{16}} \neq 0$ and $\forall i, r^{16}_{i,n_{16}+1} = 0$.

[4]This logarithmic scaling limits the number of variables and, accordingly, the amount of data the metric has to produce.

Discussion. The metrics proposed above are very difficult to interpret directly, but they discriminate execution slices against each other, with respect to temporal and spatial locality of data memory accesses. This characteristic is sufficient to apply a statistical classification method.

Moreover, it can be noticed that these metrics do not depend on the slice size: the behavior for a slice, say, twice as big would be similar to the behavior of two smaller consecutive slices. For this reason, the fixed slice size we chose here (see 4.3) does not restrict its generality.

At last, characterizing locality this way have several computational advantages: it is possible to gather several LRUDI distributions (different line sizes) for all the slices of execution in a single simulation pass. Also, this simulation is far less time consuming than simulating a couple of cache configurations or computing the real LRU distance (i.e. the number of *distinct* references between 2 references at the same (line) address) which necessitates stack processing [2, 15].

4.2 Classification Method Used

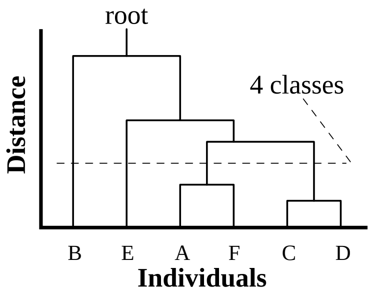

Figure 1: A classification tree.

Non-hierarchical classification methods [11] are computationally efficient, but they require an *a priori* known number of classes and an estimation of their center location. Instead, hierarchical classification methods [10] "form the final classes by hierarchically grouping subclusters or splitting parent clusters." For an aggregative (i.e. grouping) method, a hierarchical classification method derives all classification levels from a set of clusters corresponding to the set of points (each point is a cluster), to a unique cluster grouping all the points. The classical representation for a hierarchical classification result is a tree (or *dendogram*), where each level indicates the merging (or splitting) of two subclusters (see Fig. 1).

Once a hierarchical classification method is applied to a data set, each level in the tree gives a number of classes. Since only one slice (the representative point) will be simulated per class, the amount of simulated data depends on the chosen level in the tree.

We chose to use a hierarchical classification method because the entire tree is automatically generated and all the classification levels can be examined before one is chosen. Furthermore, our *wmdc* indicator can be examined for all the levels in the tree.

For this study, we have used a tool called CHAVL which implements the Likelihood Linkage Analysis (LLA) hierarchical classification method [19, 20]. This classification tool has proven to be efficient: only a few minutes were necessary to get the classification of a data set containing up to 8000 points with 61 variables. Note that other hierarchical classification tools may be suitable for this study but they may be more time and memory consuming.

4.3 Experimental Setup

We used the **calvin2** + DICE toolset [17] to run the LRUDI metrics and enable on-the-fly cache simulations. The **calvin2**+DICE toolset consists in:

- an assembly code annotator (**calvin2**) which provides target programs with an efficient fast-forwarding mode,
- and DICE, an embedded emulator which makes it possible to run on-the-fly simulations.

At run time, target programs can switch from direct execution (fast mode) to the emulation mode managed by DICE.

To simulate various level-1 data cache configurations (detailed in Table 1), we connected the Dynero IV on-the-fly cache simulator [5] to DICE in the target programs.

Varying size (from 4KB to 512KB)	Varying line size (from 16B to 128B)
4-way set associative, 32-byte line size, LRU, write back, write allocate.	4-way set associative, 32 KB, LRU, write back, write allocate.

Table 1: Level-1 data cache configurations simulated.

We tested and validated our approach on the SPEC95 benchmark suite [6]. Each program was compiled with gcc (version egcs-2.90.29), with the `-O3` optimization option, and run on a 143 MHz UltraSPARC-I workstation running Linux (UltraPenguin-1.1.9 distribution, kernel 2.2.10).

For this study, the slice size was not important because we only focussed on the choice of the slices and their representativity among all the consecutive executed slices. For this reason, we chose an arbitrary slice size: one million instructions.

4.4 Experimental Results

For each program execution slice, we collected LRUDI for line sizes of 16, 64, 256, and 4096 bytes. Experiments showed that this set of line sizes was sufficient to characterize the locality of the data reference stream with respect to the range of data cache configurations we simulated.

Besides, we simulated each SPEC95 benchmark with several level-1 data cache configurations (varying sizes and line sizes) and collected results for every execution slice from the beginning to the end. The purpose of our experiment was not to define a set of representative execution slices that would be used for several years (during the lifetime of a microprocessor design project) since this would have required using many workstations during a month or so. Instead,

[5] See `http://www.cs.wisc.edu/~markhill/DineroIV/`

[6] Instructions caches were not tested because SPEC95 benchmarks generate too few instruction cache misses [4].

our experiment only demonstrates the validity of our approach with affordable CPU time. For this reason, on a few applications, we reduced the *train* input data sets to limit the number of executed instructions to a few billions. All other programs were run with *train* input data set. Note that this does not restrict the generality of our results since we evaluated the representativeness of the selected slices among all other slices executed, and since most of the data cache configurations we simulated exhibited non-negligible miss rates.

At last, in order to validate our slice selection approach, for each simulated cache configuration, we compared the weighted mean cache miss rate computed over the selected slices [7] with the overall mean cache miss rate. To represent this comparison, we computed the relative error:

$$RE(\%) = 100 \times \frac{|\text{overall mean miss rate} - \text{weighted mean miss rate over selected slices}|}{\text{overall mean miss rate}}$$

Once, for a given target program, the classification tree has been obtained, there is still a tradeoff to consider to cut the tree. Globally and intuitively, better accuracy in the simulation results is obtained when the tree is cut at a level where many classes appear because many slices are simulated. In other words, the rate of simulated instructions over the entire executed instructions (we call it the *simulation rate*) is high. On the other hand, if the classification tree is cut near its root, a few classes appear, so only a few execution slices are selected and simulated. In this latter case, simulation take less time, but results are expected to be more erroneous.

For the SPEC95 programs, we examined several levels in the classification tree and examined the value of the *wmdc* indicator in conjunction with the *simulation rate*. The slice selection gave accurate results (i.e. relative error smaller than 10 % for absolute cache miss rates higher than 1 %) when the *wmdc* indicator was less than [8] 2 and the simulation rate was more than 0.5 %. For all the programs but `gcc`, obtaining this value of the *wmdc* necessitated to cut the classification tree at level which gave a few classes as summarized Table 2 (the maximum relative error is for cache miss rates of more than 1 %). The cache configurations simulated (column "Max. relative error" of Table 2) are those described in Table 1 (column "varying size") and the simulations on the selected slices used trained caches.

`Gcc` appears to have a characterization difficult to classify. Hopefully, our indicator allows us to be aware of this feature. So, we cannot expect to extract a few, very representative slices: we know before running the cache simulations that a simulation rate of less than 32.42 % may lead to erroneous results. However, with a simulation rate of 7.77 % (*wmdc* = 2.63), the maximum relative error for the cache simulation (for cache miss rates less than 1 %) was 5.74 % which is still acceptable despite the relative high value of the *wmdc*.

[7] The weights are those presented in Section 3.

[8] Note that this value depend on the variables used for the classification (i.e. the representation of the metrics).

	Total instruction count ($\times 10^6$)	*wmdc*	Simulation rate (%)	Max. relative error for cache sim. (%)
compress95	1183	1.84	1.01	4.60
gcc	1262	1.99	32.41	1.67
go	2315	1.95	0.56	3.11
ijpeg	1445	1.97	6.37	0.36
li	1251	0.98	0.56	1.92
m88ksim	275	0.72	0.40	4.18
perl	2595	0.03	0.50	0.09
vortex	2844	1.02	0.53	3.27
applu	444	1.59	2.03	2.72
apsi	2365	1.54	0.51	0.34
fpppp	429	0.88	0.70	1.55
hydro2d	6085	1.99	3.37	0.37
mgrid	3160	1.54	0.51	3.40
su2cor	993	1.98	1.51	3.86
swim	683	1.91	3.66	1.51
tomcatv	8000	1.30	0.51	0.08
turb3d	1743	1.77	0.80	7.41
wave5	2714	1.99	1.33	0.80

Table 2: SPEC95 slice selection results.

4.4.1 Comparison with Other Techniques

We compared our method using the CHAVL classification tool, with a method consisting in choosing one big execution slice after the initialization stage, and the systematic statistical sampling method.

In order to determine the representativeness of the simulated slices, results for the three methods correspond to cache simulations on the selected slices with trained caches: we used cache simulations on the complete run of the programs, and results were gathered each million simulated instructions.

For the "big slice" method, the resulting cache miss rates are averages upon consecutive 1-million slices representing 10 % of the applications. A program-dependent initialization phase has been skipped.

To implement the systematic statistical sampling approach, simulations are run over 1-million instruction samples. The interval between samples is a multiple of 1-million instructions semi-randomly generated to avoid periodic behavior as in [9].

Results are displayed only for the CINT95 'go' program, and the CFP95 'tomcatv' program. Both programs were chosen because they gave quite representative results and they exhibit non-negligible cache miss rates: from 2 % to 20 % (cache configurations which produced values of less than 1 % are not displayed because in this case the computed relative error does not make sense).

Figure 2 shows relative errors on CINT95 'go' for the simulations of the selected slices using our approach, the "big slice" method [9], and systematic sta-

[9] For this program, the "big slice" was taken after fast forwarding 500 million instructions,

tistical sampling. The "simulation rate" is the ratio of the simulated activity over the simulation of the complete program execution and is displayed in percent. For simulations using our approach (labeled `chavl`), the value of the *wmdc* indicator is shown in parentheses with the corresponding simulation rate. Cache configurations are those described in Table 1.

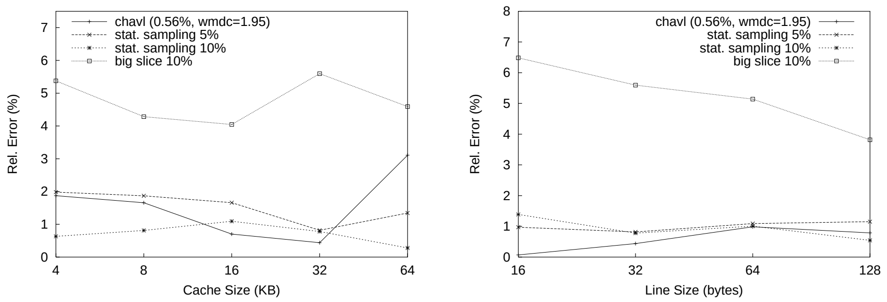

Figure 2: SPECint95 'go' data cache miss rate relative errors for various simulation methods.

Similar results for the CFP95 'tomcatv' program are displayed in Figure 3 with different simulation rates (since this program executed more than 8000 millions instructions, the "big slice" was taken after fast-forwarding 2000 million instructions).

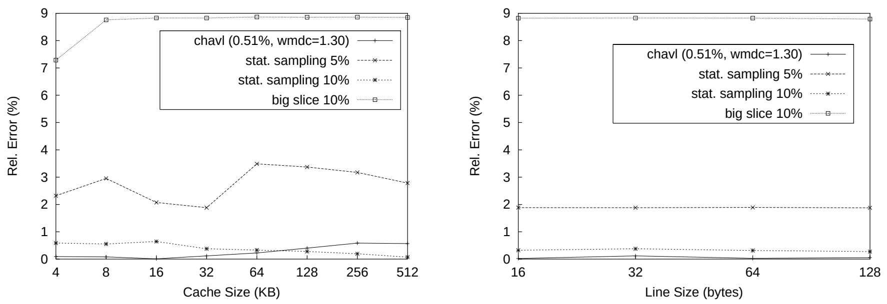

Figure 3: SPECfp95 'tomcatv' data cache miss rate relative errors for various simulation methods.

Results for other SPEC95 programs are quite similar: all the programs exhibited lower relative errors for cache miss rates using our approach than any other approach considered here with the same simulation rate.

As illustrated by `gcc` there exists applications for which a very large set of slices would be needed to be sure to get a representative behavior of the memory

since the total number of instructions executed was 2315 millions.

stream. We would like to point out that when our method does not make it possible to extract a small set of slices representative of the behavior of the programs, our indicator (*wmdc*) allows the user to be aware of the degree of representativeness of the selected slices in each class.

As noted above for `gcc`, despite a high value of the *wmdc* indicator, simulations results on the slice set as a whole may still be quite representative. A high value of the *wmdc* only indicates that we should not be too confident in the representativeness of the simulation results because the average distance of the points from their representant is high. For this reason, we are currently working to find out other indicators which, combined with the *wmdc*, may give a more accurate estimation of the representativeness of the selected slice set.

5 Summary and Conclusion

In this paper, we have presented an efficient approach to select representative slices of program execution for microarchitecture simulations. To this end, we first characterize the target program behavior using metrics chosen independent from the further simulated microarchitecture implementation. The gathered program characterization consists of measures for each consecutive fixed size slice of program execution. From this program characterization, we derive a multivariate statistical representation of the program execution (each individual being an execution slice) and then apply a classification method on it. The classification groups sets of "close" execution slices. Then, in each group, the slice nearest to the virtual center is selected (we assume it to be the most representative slice in its group) and weighted by the representativeness of the group (i.e. the number of slices in the group over the total number of slices).

We have applied this method to data memory accesses, and we have compared it with the commonly used technique consisting in choosing one big execution slice and the trace sampling approach. For cache simulations on the SPEC95 benchmarks, our method exhibited better results in terms of slice representativity. Also, to obtain results equivalent to the trace sampling method, far less program activity has to be simulated (in most cases less than 1 % is sufficient instead of 10 % for trace sampling).

In this study, we have deliberately focussed on the representativity of the selected slices and therefore reported simulation results for trained caches. However, for real cache simulations, the problem of cold start misses at the beginning of the selected slices is the same as in the trace sampling case, at the beginning of the samples. For this reason, methods applied to the trace sampling approach to reduce this "non-sampling bias" [5] (such as the use of bigger slices, warm-up periods, trace *stitching*... [1, 7, 6, 28]) may also apply to our approach.

Slices selected by our method may be far from the beginning of the program. This necessitates to skip a lot of instructions before the simulation starts as enabled by the direct execution mode of the **calvin2** + DICE toolset [17].

The ultimate use of the method presented in this paper is for complete microprocessor simulations. To this aim, we plan to add several metrics to also characterize the instruction stream (e.g. instruction mix, control transfer instruction characterization), the data dependencies (e.g. distribution of data dependency distances), etc. In the end , a classification will be applied to all the program slices, each of them being characterized by all the metrics.

Acknowledgement

The authors would like to thank Pr. Israel César Lerman from IRISA for providing us with the classification tool CHAVL and helpful discussions about statistics in general.

References

[1] A. Agarwal, M. Horowitz, and J. Hennessy. Cache performance of operating systems and multiprogramming workloads. *ACM Transactions on Computer Systems*, 6(4):393–431, November 1988.

[2] B. T. Bennett and V. J. Kruskal. LRU stack processing. *IBM Journal of Research and Development*, 19(4):353–357, July 1975.

[3] D. C. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[4] M. J. Charney and T. R. Puzak. Prefetching and memory system behavior of the SPEC95 benchmark suite. *IBM Journal of Research and Development*, 41(3), 1997. `http://www.almaden.ibm.com/journal/rd/413/charney.html`.

[5] T. M. Conte, M. A. Hirsch, and K. N. Menezes. Reducing state loss for effective trace sampling of superscalar processors. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 468–477, Washington - Brussels - Tokyo, October 1996. IEEE Computer Society.

[6] P. Crowley and J. L. Baer. On the use of trace sampling for architectural studies of desktop applications. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computing Systems (SIGMETRICS-99)*, volume 27,1 of *SIGMETRICS Performance Evaluation Review*, pages 208–209, New York, May 1–4 1999. ACM Press.

[7] P. J. Crowley and J. L. Baer. Trace sampling for desktop applications on windows NT. In *Proceedings of the Micro Workshop on Workload Characterization*, November 1998.

[8] P. K. Dubey and R. Nair. Profile-driven sampled trace generation. Technical Report RC 20041, IBM Research Division, April 1995.

[9] J. W. C. Fu and J. H. Patel. Trace driven simulation using sampled traces. In *Proceedings of the 27th Hawaii International Conference on System Sciences.*, January 1994.

[10] D. J. Hand. *Discrimination and Classification.* Wiley, 1981.

[11] J. A. Hartigan. *Clustering Algorithms.* Wiley, New York, 1975.

[12] J. L. Hennessy and D. A. Patterson. *Computer Architecture, A Quantitative Approach.* Morgan Kaufmann, 2nd edition, 1996.

[13] V. S. Iyengar, L. H. Trevillyan, and P. Bose. Representative traces for processor models with infinite cache. In *Proceedings of the Second International Symposium on High-Performance Computer Architecture*, February 1996.

[14] R. E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.

[15] Y. H. Kim, M. D. Hill, and D. A. Wood. Implementing stack simulation for highly-associative memories. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1991.

[16] H. S. Konijn. *Statistical Theory of Sample Survey Design and Analysis.* North-Holland Publishing Company, 1973.

[17] T. Lafage and A. Seznec. Combining light static code annotation and instruction-set emulation for flexible and efficient on-the-fly simulation. To appear in Euro-Par 2000, August 2000.

[18] S. Laha, J. Patel, and R. Iyer. Accurate low-cost methods for performance evaluation of cache memory systems. *IEEE Transactions on Computers*, 37(11):1325–1336, 1988.

[19] I. C. Lerman. Foundations of the likelihood linkage analysis (lla) classification method. *Applied Stochastic Models and Data Analysis*, 7:63–76, 1991.

[20] I. C. Lerman. Likelihood linkage analysis (lla) classification method: An example treated by hand. *Biochimie*, 1993.

[21] *Proceedings of the 26th Annual Symposium on Computer Architecture (ISCA)*, May 1999.

[22] *Proceedings of the 32nd Annual ACM/IEEE international symposium on microarchitecture (MICRO-32)*, November 1999.

[23] *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT).* IEEE, October 1999.

[24] T. Sherwood and B. Calder. Time varying behavior of programs. Technical Report CS99-630, University of California, August 1999.

[25] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark. Branch prediction, instruction-window size, and cache size: Performance tradeoffs and sampling techniques. *IEEE Transactions on Computers*, 48(11):1260–1281, November 1999.

[26] Alan Jay Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[27] A. Sodani and G. S. Sohi. An empirical analysis of instruction repetition. *ACM SIGPLAN Notices*, 33(11):35–45, November 1998.

[28] D. A. Wood, M. D. Hill, and R. E. Kessler. A model for estimating trace-sample miss ratios. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 1991.