RC 20610 (10/30/96)
Computer Science

IBM Research Report

Evaluation and Generation of Reduced Traces for Benchmarks

Vijay S. Iyengar, Louise H. Trevillyan

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York

IBM Research Division
Almaden · T.J. Watson · Tokyo · Zurich

Evaluation and Generation of Reduced Traces for Benchmarks

Vijay S. Iyengar   Louise H. Trevillyan

IBM Research Division
Thomas J. Watson Research Center
Yorktown Heights, NY 10598
{vijay, louise} @ watson.ibm.com

Abstract

Performance analysis using standard benchmarks is necessary to evaluate various tradeoffs during the design process. The increasing complexity of these benchmarks (e.g., from SPEC92 to SPEC95) combined with the relatively slow rate of detailed performance analysis has resulted in a widening gap between the size of the workload and the speed of analysis. This has necessitated the use of reduced traces that represent the original workload. Unfortunately, comprehensive methods of evaluating the representativeness of these reduced traces have been lacking. We present the first systematic method of evaluating such traces with respect to the combined behavior of the processor and memory subsystems. Experimental results show the usefulness of our metrics in the evaluation of any reduced trace and diagnosis of any detected deviations. We also present a method of synthesizing reduced traces which is more effective than the traditional sampling scheme for a class of benchmarks.

Keywords: dynamic traces, timer, performance evaluation, processor design.

1 Introduction

Industry standard benchmarks (e.g., SPEC95) are being put together to evaluate and compare performance of competing systems. In order to keep up with the rapid increase in the performance of systems being brought to market, these benchmarks have been growing in size. For example, the SPEC95 suite has eighteen benchmarks with each of them requiring execution of tens of billions of instructions on current systems [1, 2, 3]. Analysis tools like timers [4, 5, 6, 7] can process these traces only at the rate of tens of thousands of instructions per second or less, making the detailed analysis of the entire benchmark workload impossible by such software tools. This has led to the need for having, for each benchmark, reduced traces that are representative of its workload.

The first step is to define the notion of *representativeness*. Consider two systems A and B that are being evaluated using a benchmark. A reduced trace for this benchmark can be used for this evaluation if both A and B have the same behavior on the reduced trace as they would on the full trace. In other words, the usages and transitions between the states of each of these systems should be proportionally similar for the full and reduced traces. Unfortunately, it is not practical to model the entire state of a system to verify the representativeness of a reduced trace. The main result of this paper is to present a systematic method of evaluating representativeness using a simplified model of the system (processor + memory). The challenge is to include, either implictly or explicitly, all the important parts of the target system in the model without making the analysis unmanageable. Our evaluation method is illustrated by experiments using some of the SPEC95 benchmarks. We also present a scheme for generating representative reduced traces for any benchmark. Experimental results indicate that for some of the benchmarks our reduced traces track the full workload better than those produced by earlier schemes.

2 Background

The generation and characterization of reduced traces have been studied quite extensively for the memory subsystem. For example, recent work [8] compares cache behavior of sampled traces with that of the original full trace. Characteristics compared include cumulative and instantaneous cache miss rates and inter-miss distances. Methods of handling the errors due to discontinuities at the sampling boundaries are also suggested.

Quantifying representativeness of reduced traces with respect to processor behavior has not received much attention. Simple metrics such as instruction frequencies, basic block length and usage frequencies have been used [6]. However, such simple metrics are not discriminating enough to quantify performance impact due to factors like context. More recently, we presented a more rigorous metric that handled key features like branch prediction, register interlocks, and contention for resources [9]. However, this work was restricted to infinite cache models and left unanswered the question of how to handle both the processor and memory subsystems when trying to quantify representativeness.

In this paper, we present a framework for using the combined behaviors of the processor and memory subsystems to characterize the representativeness of any reduced trace. Computer systems being designed today attempt to achieve higher performance by a careful balance between the processor and memory subsystems, making it important to consider interactions between these two subsystems. For example, the effectiveness of a memory latency hiding technique in a processor can be adequately evaluated only if the combined processor and memory subsystem behaviors are analyzed. Considering the behavior of these two subsystems independently in any metric would be an oversimplification.

The need for reduced traces has led to many schemes being proposed to generate reduced traces with varying amounts of success. The first set of methods use some variation on sampling to shrink the full trace down to a manageable size [5, 7]. As we will show later in this paper, the quality of the resulting trace varies with the benchmark. The specific complexities of a benchmark make it hard to devise a general sampling scheme that produces a representative reduced trace for all workloads. A more recent method uses one pass over the trace to collect basic block usage statistics and another pass to emit basic blocks when needed to meet their scaled target usages [10]. The basic block usage statistics collected in this scheme ensure that reduced trace has the blocks in the right proportion. However, there is no guarantee that the *right* instances of the basic blocks were chosen. The notion of *right* can be quite complex depending on details of the system. This will be apparent in the later discussion on our metric. Lastly, one can try to alter the input to the benchmark to generate a reduced trace that is representative of the original workload. This manual process requires a deep understanding of the workload and its impact on the processor and memory subsystems. The ability to impact the behavior of instruction and data caches, branch prediction logic etc. simply by altering the inputs depends greatly on the benchmark.

In this paper, we also present a reduced trace generation scheme that uses the combined characterization of the processor and memory subsystems needed to compute our metric. The generation heuristic attempts to minimize the metric using models of key microarchitectural features of the design being targeted. Hence, unlike the earlier generation schemes, this method produces reduced traces targeted for specific designs.

3 A New Metric

The metric (called R-metric; R denotes representativeness) will be developed in a two step process. In an earlier work [9] we defined the R-metric for infinite cache systems. In this work we extend the metric to include components from both the processor and memory subsystems. As mentioned earlier, the challenge in the development of this new metric is to balance the need to model key details of the system against the explosion in the collected statistics for large industrial benchmarks.

In our methodology the characterization is tailored to the target system that is being analyzed. In this paper, we will illustrate the methodology using a system that contains a superscalar processor that can have up to 16 outstanding instructions. It contains some combination of fixed point units, floating point units, load/store units, and a branch unit. A 2-bit saturating counter is used for branch prediction and the branch history table has 512 entries. The design does not have any data prefetching mechanisms. Separate instruction and data TLBs are 2-way set associative and contain 128 entries each. Separate instruction and data first level 16KB caches are organized as 4-way set associative. The line size is 32 bytes for both. The size of the direct mapped unified second level cache is 1MB with a line size of 32 bytes. To apply our methodology to another design, many of the parameters described above will have to be changed in the characterization (e.g., cache size and organization). Some new features may require addition of a new functional unit to the characterization. For example, a design with a data prefetching unit would require that the characterization include some way of modeling the predictability of data addresses.

3.1 Qualifying Single Instructions

As mentioned earlier, the reason why simple instruction usage statistics prove inadequate is because they ignore the different conditions under which the same instruction gets executed at different points in the trace. We provide a way to make such distinctions by qualifying the execution characteristics of each instruction in a systematic fashion. In the first step, we consider ways of qualifying single instructions. Clearly, the qualification will depend on the type of instruction. However, one major functional unit is common to all instructions, namely, instruction fetch.

The instruction fetch operation is affected by whether or not it encounters a hit or a miss in the instruction TLB, the level one instruction cache and, if needed, in the second level cache. Each instruction will be qualified by three bits denoting hit/miss in the instruction TLB and the first and second level caches. Not all combinations of these three bits are relevant. For example, the state of second level cache is not relevant when there is a hit in the first level cache.

Branch instructions are also qualified based on the outcome of the branch prediction logic. This outcome can have one of three possible states: branch is predicted correctly, branch is mis-predicted, no prediction for this branch. The last category is needed because not all branch types are handled by the prediction logic. Our choice of qualifying using the branch prediction outcome rather than the state of the underlying branch prediction counter results in fewer, but all the relevant, qualified entities. Different branch prediction schemes (e.g., the two-level scheme in [11]) can be easily handled by changing the underlying model of the prediction scheme, but keeping the same three possible outcomes.

Load/store instructions are also qualified based on the hit/miss status in the data TLB and caches. Since a single instruction can access more than one line in the caches, the number of lines accessed in the first level data cache is used to qualify load/store instructions. Detailed hit/miss information (similar to the instruction fetch case) is modeled only for the first two of these accesses. For the benchmarks being considered this simplification does not have a significant impact since the access of more than two lines is not common.

3.2 Qualifying Instruction Context

The second step in the qualification process is to include the context. Situations like contention for resources between instructions, register dependencies and use of bypasses impact the program execution and need to be modeled. Instead of modeling such scenarios explicitly we model them implicitly by considering some number of preceding instructions as context. The number of preceding instructions included in the context should be comparable to the number of instructions that can be outstanding. For the target design used in this paper the previous 15 instructions are considered as context. Note that instructions in the context are also singly qualified as described in the previous subsection. This implies that our characterization distinguishes between two instances of an instruction that have the same preceding instructions except that in one case a preceding instruction missed in a cache while in the other case it did not. The intuition is that the cache miss of the preceding instruction would impact the execution of current instruction and requires that these two cases be differentiated. For some designs, the context must include instructions preceding and following the instruction being qualified. This is feasible in our framework, but we have not implemented this capability.

The other part of the context that is used in the qualification is related to reuse of data operands between instructions in a sequence. For example, a store to some memory location followed by a load from the same location may use some special paths and buffers specifically designed to speed this situation. In general, one would need to consider operand lengths and partial reuse. We have implemented a simplified version of interlock modeling where symbolic values are used to represent the data operand addresses in the current instruction and the context. The symbolic addresses indicate equality/inequality relationships between these data addresses.

We will use the term *entity* to refer to an instruction fully qualified as described above. Consider the entity displayed in Table 1 that was encountered in the characterization of one of the benchmarks. The first column lists the instruction type (load/store, branch and other) and its location. The current instruction is listed first with the context following in order (oldest last). Instruction TLB and first and second level cache status are given next for each instruction. The notation used for TLB and cache status is h for hit, m for miss and * to indicate not relevant. These are listed in the order (TLB, first level cache, second level cache) for both instructions and data accesses. For each load/store instruction, the number of data lines accessed and the status for the first two accesses are shown (e.g., instruction at 240). Symbolic data addresses indicate that there is no operand reuse in this example of an entity.

Instruction	Instruction TLB and caches	Branch prediction	Symbolic data addresses	Data TLB and caches		
				Accesses	Status-1	Status-2
L/ST at 264	hh*		A	1	hmm	-
Other at 260	hmm					
L/ST at 25c	hh*		B	1	hh*	-
Other at 258	hh*					
L/ST at 254	hh*		C	1	hh*	-
Other at 250	hh*					
Other at 24c	hh*					
Other at 248	hh*					
Other at 244	hh*					
L/ST at 240	hmm		D	3	hmm	hmm
Branch at 1f0	hh*	not predicted				
L/ST at 1ec	hh*		E	1	mmm	-
L/ST at 1e8	hh*		F	1	hh*	-
L/ST at 1e4	hh*		G	1	hh*	-
L/ST at 1e0	hmm		H	1	hh*	-
L/ST at 1dc	hh*		I	1	hh*	-

Table 1: Example of an entity

3.3 Computing the Metric

The two-step qualification process can be used to analyze any trace and determine the characteristics for each instruction in the execution trace. We will use the notation Ω to denote a entity. The formula for computing the R-metric is similar to that introduced in [9], except that it has to be applied to these new entities that are instructions qualified by both processor and memory behaviors.

Consider an original program trace T that contains $\mid T \mid$ instructions and a reduced trace τ for the same program containing $\mid \tau \mid$ instructions. Let $count_T(\Omega)$ denote the number of occurrences in T of each entity Ω. Let $count_\tau(\Omega)$ denote the number of occurrences of the same entity Ω in τ. Let

$$scaled_{T,\tau}(\Omega) = \frac{count_T(\Omega) \times \mid \tau \mid}{\mid T \mid} \tag{1}$$

denote the expected scaled count for Ω in a representative trace of size $\mid \tau \mid$.

$$\begin{aligned} R = \quad & \tfrac{1}{|\tau|} \times \textstyle\sum_\Omega (scaled_{T,\tau}(\Omega) - count_\tau(\Omega)) \\ & \forall\ entities\ \Omega\ such\ that\ (scaled_{T,\tau}(\Omega) - count_\tau(\Omega)) \geq 0 \end{aligned} \tag{2}$$

The R-metric defined in Equation 2 measures the deviation in the reduced trace from the ideal scaled counts for each entity. This is expressed as the fraction of instructions in the reduced trace that do not have the desired characteristics. An ideal reduced trace will have a R-metric value of 0, but this is not always an achievable target.

Notice that the R-metric calculation penalizes all deviations from the ideal qualification equally. For example, a deviation in cache behavior of the oldest instruction in the context is penalized the same amount as the same deviation in the cache behavior of the current instruction. One could expect the same deviation to have more impact on the execution of an instruction when it occurs closer to it in the instruction sequence. Consider another example where the deviation in the qualification of the current instruction involves both the TLB and the first level cache in one case and only the first level cache in the other case. Again, one could expect the first case to have a bigger impact on performance compared to the second case. These cases could be addressed by weighting each deviation based on the estimated performance impact. This work is ongoing and all the results in this paper will use the simpler metric calculations in Equation 2.

In order to compute the R-metric for a reduced trace we first need to characterize the full trace by identifying the qualified instructions present in it and their usages. The memory and run time for this pass over long traces is determined to some extent by the complexity of the qualification used. A finer qualification that distinguishes more instances of the same instruction will result in more entities being found and also more time spent identifying them. For the characterization described in this section we are able to analyze about 100K instructions per second on a modern workstation. This implies that a benchmark with about 20B instructions will be characterized in about 2-3 days. Once the full trace is characterized using the appropriate parameters for the target design, any reduced trace for that benchmark can be analyzed and the metric computed at the 100K instructions per second rate. Section 5 will present results of evaluating reduced traces for some of the SPEC95 benchmarks.

4 Reduced Trace Generation

The intuition behind the R-metric that was developed in the previous section is that a representative reduced trace should preserve the proportion of instructions with various characteristics seen in the full trace. This leads us to second part of this work, i.e., generating representative reduced traces by attempting to minimize the corresponding R-metric. It was shown in [9] that this problem is hard. Also, the large problem sizes encountered force the use of simple heuristics. The experimental results in the next section will show that these heuristics are successful for a class of benchmarks.

The algorithm presented in Figure 1 uses the information on entities and their usages collected in the full trace characterization described in Section 3. In addition, counts of transitions between basic blocks are needed for our generator. The generator uses the basic block transition information to build a directed graph G with basic blocks as nodes V and edges E representing all the transitions seen in the full trace.

The graph is first decomposed into its strongly connected components (line 1 in Figure 1). Since the graph is generated from a single trace of program execution the strongly connected components will form a linear chain. The basic block transition counts are scaled and rounded for the target reduced trace size (line 2). The reduced trace is generated incrementally by traversing the graph starting from the first strongly connected component to the last (lines 3-4). Exactly the same models of the design features used in the full trace characterization are used and updated to reflect the choices made in the partially generated trace at every step of the generation process. During the graph traversal, at any node (i.e., basic block) the unused edge counts are used to determine choices for the successor node (line 6). For each of these choices the first instruction of the basic block represented by this successor node is characterized to determine the resultant entity Ω (line 9). This particular entity Ω may or may not have been seen in the full trace. If not, a distance is computed to the closest entity that was seen in the full trace (line 10). This notion of distance penalizes deviations closer to the current instruction more than those in the older context. Also, two deviations in the same position may be penalized differently based on a rough estimate of their relative impact on performance (e.g., the penalty for a TLB miss is usually higher than that for a first level cache miss). The successor node is determined by picking the smallest distance choice which still has target usage left (line 11). Note that any choice of a successor with a non-zero distance would result in the metric getting worse. This greedy approach using distances tries to recover from non-zero distance choices by picking nodes that would tend to reduce the distance.

For each node picked, the instructions are added to the reduced trace being generated. For load/store

```
program GENERATOR (G(V, E))
1 Decompose G into its strongly connected components (SCC's)
2 Determine scaled and rounded target usage for each SCC_i and for each node v ∈ V
3 For each SCC in linear order starting with SCC_0
4       While target usage for SCC_i is not met
5         v = current node in SCC_i, starting with entry node of SCC_i
6         C = children of v with target usage remaining
7         If C ≠ φ
8            then For each node w ∈ C
9                       Consider the resultant qualification Ω for first instruction i ∈ w
10                      Compute minimum distance D_w of Ω to any qualification of i with target usage left
11                 Pick next node v' = w with minimum distance D_w
12           else Pick next node v' ∈ SCC_i with remaining target usage
13                and shortest path to v in G
14        For each instruction i ∈ v' in sequential order
15             If type(i) = load/store
16                then Pick a qualified version of i, say Ω, with remaining target usage
17                         and minimum distance to current entity
18                      Generate data address for i with memory characteristics extracted from Ω
19                      If generation fails retry lines 16-18 with new Ω using non-decreasing distance order
20             Add i to reduced trace being generated
end GENERATOR
```

Figure 1: Sketch of generation algorithm

instructions data addresses have to be generated before they are emitted to the reduced trace (lines 15-18). This is done by examining for each load/store instruction in the basic block all of its qualified versions seen in the full trace. We chose the qualified version, say Ω, that has minimum distance with respect to the context of the current load/store instruction in the reduced trace and also has target usage remaining (lines 16-17). The memory subsystem characteristic for the current load/store instruction is extracted from the chosen qualified entity Ω and a data operand (i.e., starting address and length) generated to satisfy this characteristic (line 18). The generation of data addresses with specified characteristics is a non-trivial problem since it must be accomplished with the given memory subsystem state. In some cases, it may be impossible to achieve the requested characteristic given the state of the caches and TLB. In this case, a different Ω is chosen in order of non-decreasing distance and the generation attempted again (line 19). If all such attempts fail, the default characteristics of misses in all caches and TLB can always be satisfied.

The data addresses seen in the full trace are used in the beginning to seed the data address generation process. However, as the address generation process continues the sequence of generated addresses bear no resemblance to those seen in the full trace. The advantage with our synthetic data addresses is that they are chosen to satisfy the constraints based on desired memory characteristics. If the generation is largely successful, the data addresses will have the desired characteristic irrespective of the size of the reduced trace. This is to be contrasted with traces generated, say, by sampling, where one may need to keep the reduced trace size relatively large just to achieve the right cache miss/hit rates. A disadvantage with synthetic data addresses is that they are chosen solely based on modeled memory behavior. If some key part of the memory model were omitted one would not expect any accidental conformance with respect to it. For example, if data prefetching is relevant but not modeled then the generated data addresses would not reflect the proper prefetch behavior.

Lines 12-13 of the algorithm in Figure 1 are needed to pick a successor basic block if none of the children are suitable candidates. This situation can arise if target transition counts between the basic blocks do not satisfy the criterion for an Euler traversal of the directed graph G [12]. The problem of finding integer transition counts closest to the ideal scaled counts (say, using difference squared criterion) is hard to solve especially for the large graphs we tend to encounter. Our experience is that results are generally better for transition counts generated by rounding the ideal scaled counts even if they do not satisfy the criterion for an Euler traversal. However, some timer models may require input traces to conform to some Euler traversal of the basic block graph.

5 Experimental Results

In this section we will present results for some benchmarks from ongoing experiments to illustrate the uses of the R-metric in evaluating various reduced traces. Reduced traces generated by the algorithm described in Section 4 will also be evaluated using our metrics.

The benchmarks chosen for analysis in this paper are from the SPEC95 suite. We picked two floating point benchmarks with different characteristics: tomcatv and wave5. From the integer suite of SPEC95 we picked perl which has two inputs: scrabbl.in and primes.in. We have chosen to characterize these two inputs separately in order to do some comparisons and they will be denoted perl-s and perl-p, respectively. We have also chosen gcc (actually cc1 since our tracing tool does not handle forks) but restricted it to just one of its many inputs: decl. This will be denoted by cc1-decl. The inclusion of gcc does not need any justification since it is well known to have many characteristics that stress system designs.

Characteristic	tomcatv	wave5	cc1-decl	perl-s	perl-p
No. of instructions in trace	20.16B	23.33B	1.26B	29.66B	15.71B
Code size (in instructions)	7953	22611	187156	21167	15549
No. of unique singly qualified instructions	9229	26681	277339	26993	19165
No. of unique entities (incl. context)	34505	758748	800037*	488569	94830
D-TLB miss rate	0.08%	0.93%	0.02%	0.01%	0%
D-L1 miss rate	14.9%	16.6%	1.01%	0.56%	0%
D-L2 miss rate	6.18%	1.14%	0.06%	0.01%	0%
I-TLB miss rate	0%	0%	0.02%	0%	0%
I-L1 miss rate	0.02%	0.02%	2.28%	0.83%	0%
I-L2 miss rate	0%	0%	0.06%	0%	0%

Table 2: Key statistics from characterization runs on benchmarks

Each of these benchmarks was characterized using the design parameters and techniques described in Section 3. Key results from this characterization are given in Table 2. First, for each benchmark the table gives the number of instructions encountered in the trace (i.e., size of the trace). Next, the size of the code is given indicating the wide range in instruction footprint size from tomcatv to cc1-decl. Note that only instructions that were executed at least once (i.e., appear at least once in the trace) are included in the code size. The first step of the qualification process described in Section 3.1 considered single instructions in combination with key states in the processor and memory subsystems. Hence, an instruction at a particular location in the code may appear in different incarnations due to this qualification. The number of singly qualified instructions exceeds the code size by 16% in tomcatv and by 48% in cc1-decl. Inclusion of the context also in the entities results in more dramatic increases to the numbers of qualified entities. When compared to the code size these increase ranges from a factor of around 4 for tomcatv to a factor of around 34 for wave5.

We did not compute the true count of entities (including context) for the cc1-decl benchmark. The richness of the characterization being done would result in a very large number of entities. However, this does not mean that we cannot apply our techniques to difficult benchmarks like cc1-decl. The solution to this explosion problem lies in the observation that most of the reference trace content can be captured by a subset of the qualified entities. For the cc1-decl the characterization run was done with a preset limit of around 800K qualified entities (indicated by * in Table 2). When the number exceeds this limit entities that were not used very often are discarded in favor of entities with potential for more use. The success of this technique is indicated by the fact that the roughly 800K entities characterized contained 94% of the content of the 1.26B reference trace. The benchmark cc1-decl was intentionally run in this limit mode, even though our system can handle more than a million qualified entities. As one makes finer and finer distinctions in the qualification, some kind of limit mode analysis will be necessary. The experience with cc1-decl shows promise and extensibility.

Table 2 also gives the cache and TLB miss rates (per instruction) for all the benchmarks. Only cc1-decl and perl-s stress the instruction cache to some extent. The floating point benchmarks, tomcatv and wave5, stress the data caches. The statistics shown in Table 2 reveal key characteristics of these benchmarks. For example, the workload in benchmarks with large number of entities will be harder to represent with small reduced traces. In this respect, wave5 will be harder to compress than the other floating point benchmark tomcatv. The main cause of the large number of qualified entities in wave5 is the richness in the context. A similar comparison can be made between perl-s and perl-p, which represent the same program driven by two different inputs.

	tomcatv	wave5	cc1-decl	perl-s	perl-p
R-metric (complete characterization)	8.75%	16.47%	28.79% *	3.76%	6.09%
R-metric (branch prediction only)	3.89%	12.14%	20.27% *	2.48%	4.47%
R-metric (data address interlock only)	3.85%	12.03%	19.21% *	2.04%	3.16%
R-metric (instruction TLB and caches only)	3.86%	12.17%	20.24% *	2.35%	3.19%
R-metric (data TLB and caches only)	8.70%	16.26%	20.56% *	2.88%	4.75%
R-metric (16-grams of unqualified instructions only)	3.85%	12.03%	19.21% *	2.03%	3.16%
D-TLB miss rate	0.09%	0.46%	0.04%	0.01%	0%
D-L1 miss rate	15.4%	16.2%	1.07%	0.58%	0.02%
D-L2 miss rate	6.43%	1.47%	0.32%	0.03%	0.01%
I-TLB miss rate	0%	0%	0.03%	0%	0%
I-L1 miss rate	0.03%	0.05%	2.38%	0.87%	0.09%
I-L2 miss rate	0%	0.03%	0.24%	0.02%	0.02%

Table 3: Evaluation of sampled traces of length 10M instructions

Next, we will evaluate reduced traces produced by the widely used uniform sampling method. Each sampled trace will contain 10M instructions extracted from the full trace using 50 equispaced samples of size 200K instructions. This number and size of samples is similar to that used in the literature [8, 13]. The results of the evaluation are displayed in Table 3. The top row for each benchmark gives the R-metric with the complete characterization as described in Section 3. For example, for tomcatv 8.75% of the sampled trace was not representative, considering the full characterization. The table also gives the values for the R-metric if the characterization had been done using only one of key functions like branch prediction. The R-metric value is also computed considering instructions and the context of 15 preceding instructions, all unqualified by the state of any key function (i.e., just 16-grams of unqualified instructions). Having the right sequence of unqualified instructions is clearly a prerequisite to getting low values for the other R-metrics. One should not expect the R-metric values for the individual functions to add up the R-metric value for the complete characterization since the metric penalizes all the deviations and their combinations equally. The table also gives overall TLB and cache statistics that can compared with those in Table 2 for the full trace.

For tomcatv, the data TLB and cache behavior is the dominant deviation in the sampled trace. Comparison of the overall data TLB and cache statistics does not give enough information on the deviation. However, examining qualified instructions with the large errors gives insights into what is happening. Consider the load/store instruction at location 193c. Qualifying the occurences of this instruction in the full and reduced trace using only the data TLB and caches results in identification of many qualified entities. Two of these entities, Ω_1 and Ω_2, are shown in Table 4. The instruction being qualified at 193c is shown at the top and then the context is listed with the most recent one first. Each load/store instruction in Ω_1 and Ω_2 makes only one access to the first level cache and the corresponding hit/miss status in the data TLB, level one cache and the second level cache (in order) is given in Table 4. The notation used is h for hit, m for miss and * to indicate not relevant. The only difference in these two entities is the second level cache status for an instruction at 1958 in the context. The qualification data we collected suggests that the sampled trace had only about 15% of the ideally required number of Ω_1, having instead an excess of Ω_2. This kind of detailed analysis would not be possible without the qualification done for the R-metric.

Instruction	Ω_1	Ω_2
L/ST at 193c	hh*	hh*
L/ST at 1938	hh*	hh*
Other at 1934		
L/ST at 1930	hh*	hh*
L/ST at 192c	hh*	hh*
Branch at 196c		
L/ST at 1968	hh*	hh*
L/ST at 1964	hmm	hmm
L/ST at 1960	hh*	hh*
Other at 195c		
L/ST at 1958	hmh	hmm
L/ST at 1954	hh*	hh*
Other at 1950		
L/ST at 194c	hh*	hh*
L/ST at 1948	hh*	hh*
L/ST at 1944	hh*	hh*

Table 4: Two data TLB and cache qualified instructions in tomcatv

For cc1-decl, since the qualification was done in limit mode there is missing information about some of the entities seen in the full trace. Conservatively, one would have to assume that the proportion of the reduced trace that corresponds to uncharacterized full trace is not representative. This adds 6% to the R-metric value for cc1-decl resulting in the reported value of 28.79%. Even without this additional factor the high R-metric value for cc1-decl's sampled trace indicates a lot of deviation in the qualified entities. In limit mode, the other metric values are also conservative, but the high value for the unqualified 16-grams suggests that not getting the right sequence of instructions is a big factor in the poor overall metric.

The results in Table 3 indicate that sampling does not always produce representative traces. For such cases increasing the size of the sampled trace may result in acceptable reduced traces. Such possibilities can be easily explored using the R-metric.

Another way of generating reduced traces is to change the input of a benchmark such that it shortens the execution in a representative fashion. In general, this is quite difficult to do. For example, consider the training input for wave5 (available from SPEC) as a way to shrink this workload. We evaluated this shortened execution of wave5 and the results are in Table 5. The shortened execution had 1.54B instructions. The R-metric is quite poor in spite of the large number of instructions in this reduced trace. While the data L1 miss rate shows some deviation from the full trace values, the poor value of the metric for unqualified 16-grams indicates the main reason for the poor results.

Lastly, we will evaluate the reduced traces generated using our method as outlined in the previous section. We will use the same target size of 10M instructions for these traces that was used for the sampled traces. Table 6 summarizes the results of the evaluation. The reduced traces generated using our algorithm (Section 4) for the floating point benchmarks have much better metrics than the sampled traces. The success of the synthetic data address generation is apparent by comparing the overall data cache miss ratios and the data TLB and cache qualified R-metric values. On the other hand, the results are worse for cc1-decl and perl-s benchmarks. Our generator does not seem to be very successful in getting the right sequence of instructions to satisfy the required unqualified 16-grams, the branch prediction behavior and the instruction caches/TLB behaviors. The greedy algorithm (without any backtracking) is inadequate to find the appropriate traversal of the graph such that all these behaviors track what was seen in the full trace. When the complexity in the benchmark is not due to intricate sequencing but richness in combinations of data cache behavior (e.g., wave5) the algorithm does well.

	wave5 (training input) Trace size = 1.54B instructions
R-metric (complete characterization)	26.89%
R-metric (branch prediction only)	23.59%
R-metric (data address interlock only)	23.74%
R-metric (instruction TLB and caches only)	23.62%
R-metric (data TLB and caches only)	26.45%
R-metric (16-grams of unqualified instructions)	23.53%
D-TLB miss rate	0.42%
D-L1 miss rate	13.1%
D-L2 miss rate	0.60%
I-TLB miss rate	0%
I-L1 miss rate	0.03%
I-L2 miss rate	0%

Table 5: Evaluation of shortened execution using the training input for wave5

	tomcatv	wave5	cc1-decl	perl-s	perl-p
R-metric (complete characterization)	1.47%	4.61%	36.39% *	20.33%	6.22%
R-metric (branch prediction only)	0.24%	0.20%	22.50% *	12.61%	5.38%
R-metric (data address interlock only)	0.75%	3.41%	15.33% *	2.61%	0.72%
R-metric (instruction TLB and caches only)	0.41%	0.18%	23.18% *	10.48%	0.82%
R-metric (data TLB and caches only)	0.18%	1.21%	16.51% *	4.80%	1.44%
R-metric (16-grams of unqualified instructions only)	0.01%	0.04%	15.29% *	2.13%	0.70%
D-TLB miss rate	0.09%	0.94%	0.04%	0.02%	0%
D-L1 miss rate	14.9%	16.6%	1.07%	1.05%	0.03%
D-L2 miss rate	6.18%	1.17%	0.32%	0.08%	0%
I-TLB miss rate	0%	0%	0.03%	0.14%	0%
I-L1 miss rate	0.05%	0.03%	2.38%	2.11%	0.01%
I-L2 miss rate	0%	0.01%	0.24%	0.24%	0%

Table 6: Evaluation of reduced traces generated using algorithm in Section 4

None of the generation schemes evaluated are successful in producing a representative reduced trace for cc1-decl. This problem remains unsolved. However, our characterization and metrics are able to expose this problem and also provide detailed information on the relative contributions to the deviation from different functions in the system.

6 Conclusion

Reduced traces for performance benchmarks are widely used for analyzing design tradeoffs. However, there is no agreed method to determine if these reduced traces represent the original workload in the benchmarks.

In this paper we have presented a framework to evaluate the representativeness of a reduced trace and its suitability for analyzing the performance of a target system. This framework is flexible in allowing both explicit and implicit modeling of key features in the processor and memory subsystems. The computed R-metrics are therefore very discriminating in assessing the representativeness of the reduced trace for the target system. Experimental results using some of the SPEC95 benchmarks illustrated the use of the R-metric in evaluating reduced traces produced using various methods. This analysis also produced diagnostic information on the nature of deviations when the reduced traces were not representative. We also presented a new reduced trace generation algorithm that performs better than the traditional sampled traces for the floating point benchmarks evaluated. The evaluations with the R-metrics show that the problem of producing representative reduced traces is still unsolved for some benchmarks like gcc.

References

[1] B. Case. Spec95 retires spec92. *Microprocessor Report*, August 21 1995.

[2] K. M. Dixit. New CPU benchmark suites from SPEC. In *Digest of papers, SPRING COMPCON*, pages 305–310, 1992.

[3] O. Sharp and D. F. Bacon. Measure for measure. *Byte*, pages 65–72, October 1994.

[4] Jin-Chin Chung. Modeling and evaluation of a superscalar architecture. In *Proceedings of MASCOTS'93, International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 50–53, 1993.

[5] G. Lauterbach. Accelerating architectural simulation by parallel execution of trace samples. In *Proceedings of Twenty Seventh Annual Hawaii International Conference on System Sciences*, pages 205–210, 1994.

[6] A. Poursepanj, D. Ogden, B. Burgess, S. Gary, C. Dietz, D. Lee, S. Surya, and M. Peters. The powerpc 603 microprocessor: Performance analysis and design trade-offs. In *Digest of papers, SPRING COMPCON*, pages 316–323, 1994.

[7] A. Poursepanj. The powerpc performance modeling methodology. *Communications of the ACM*, pages 47–55, June 1994.

[8] J.W.C. Fu and J.H. Patel. Trace driven simulation using sampled traces. In *Proceedings of Twenty Seventh Annual Hawaii International Conference on System Sciences*, pages 211–220, 1994.

[9] V.S. Iyengar, L.H. Trevillyan, and P. Bose. Representative traces for processor models with infinite cache. In *Proceedings of Second International Symposium on High Performance Computer Architecture*, pages 62–72, 1996.

[10] P.K. Dubey and R. Nair. Profile driven generation of trace samples. In *Proceedings of International Conference on Computer Design*, pages 217–224, 1996.

[11] T.-Y. Yeh and Y.N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 257–266, May 1993.

[12] N. Deo. *Graph Theory with Applications to Engineering and Computer Science.* Prentice Hall, Inc., 1974.

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