Accuracy and Speed–Up of Parallel Trace–Driven Architectural Simulation

IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598

Abstract

Trace–driven simulation continues to be one of the main evaluation methods in the design of high performance processor–memory sub–systems. In this paper, we examine the varying speed–up opportunities available by processing a given trace in parallel on an IBM SP–2 machine. We also develop a simple, yet effective method of correcting for cold–start cache miss errors, by the use of overlapped trace chunks. We then report selected experimental results to validate our expectations. We show that it is possible to achieve near–perfect speed–up without loss of accuracy. Next, in order to achieve further reduction in simulation cost, we combine uniform sampling methods with parallel trace processing with a slight loss of accuracy for finite–cache timer runs. We then show that by using warm–start sequences from preceding trace chunks, it is possible to reduce the errors back to acceptable bounds.

1. Introduction

The ever–increasing sizes of real workloads is making the use of trace–driven simulation methods impractical in time–bound processor development projects. In this paper, we describe a simple method of speeding up trace–driven architectural simulation through the use of parallel processing. We also address the problem of correcting for cold–cache errors. We propose an easy–to–implement method of using “overlapped” trace segments to reduce such errors. We investigate speed versus accuracy trade–offs in the context of large technical and commercial workloads. We present additional experimental results to demonstrate the worth (and risk) of using trace sampling methods in conjunction with the basic parallel trace–processing paradigm.

Prior attempts in reducing architectural simulation cost have been largely limited to trace size compaction methods. The size of the trace can be reduced through sampling (e.g. [1–5]) or synthetic regeneration [6]. In [1], parallel processing on a cluster of workstations was applied to collected trace samples. However, the process of collecting trace samples was reported to be time–consuming and the accuracy of workload metrics (e.g. basic block frequencies) was stated to be low. Also, warm–start correction for short samples was effected by explicit saving and communication of cache states. This required the original simulation toolset to be modified in source code. Our methodology (PARSIM) as explained in detail in [10], is non–invasive, in that the existing tools can be ported over without modification.

2. Parallel trace processing

In this section, we summarize the basic rationale behind the implementation of our proposed methodology. The parallel hardware platform of choice was the IBM SP–2 machine [7]: a scalable, general purpose parallel system based on a distributed memory message–passing architecture. Each node of an SP–2 system is a POWER2 super scalar processor [8]. Detailed analysis of alternate implementation modes of the PARSIM facility is available in the full technical report [10].

The basic implementation paradigm for which we report results in this paper is as follows: each processor reads and simulates its assigned trace chunk from a shared trace file. At the end of processing, each processor writes its file of statistics to the shared filesystem. A post–processing shell script applies an appropriate combining formula [10] to compute the net statistics file. If the processors consume disjoint trace chunks, (Figure 1a), there is no need, per se for inter–process communication during the analysis phase. The basic method we use to recreate simulation state (if and where needed), is to use overlapping trace chunks, (Figure 1b). Thus, for example, if the analysis program is a cache simulator, the ith trace chunk (1 < i ≤ n) starts with an overlap with the (i–1)th and so the cache is warmed up suitably to reflect the end–of–chunk state for the ith chunk. The amount of overlap required to accurately recreate the state, is a function of the cache size and the characteristics of the particular workload, as discussed next.

* A. Nguyen has now returned to University of Illinois
+ M. Michael has now returned to University of Rochester
In the rest of this paper, we will refer to simple analysis tools (for which no prior trace context is required to determine the next action) to be zero–state tools and all other simulation tools (like cache simulators or processor timers) to be finite–state tools. For zero–state tools, the error in the final computed statistics will clearly be zero, irrespective of the number of trace chunks (or processors). Hence, no “overlap” correction to eliminate cold–state errors is needed.

2.1 Trace chunk overlap sizes

For a given finite state tool, let \( W = \text{average value of prior trace length needed to determine the next processing action without error} \). Then, a reasonable choice of the trace chunk overlap size (Figure 1b) in our warm–start correction experiments, would be \( W \) instructions.

Let us consider a general instruction trace with the instructions numbered sequentially from 1 to \( N \). Each instruction may either be a memory reference instruction (load or store), or a processor op. We consider such a trace in conjunction with a cache simulation program.

Let \( C = \text{cache size in bytes} \), \( A = \text{set associativity of the cache} \), \( L = \text{linesize in bytes} \). (So, the number of “congruence classes” \( c = C/(L . A) \)). Also, let \( m = \text{workload miss ratio} = (\text{#misses}) / (\text{#memory accesses}) \) and \( r = \text{memory reference ratio} = (\text{#memory references}) / (\text{#instructions}) \).

In the above, \( m \) is the average miss ratio measured after the cache has been primed; this is the same as the average number of cache line replacements per memory access, after the initial “fill” period. We wish to estimate the average (expected) value of \( W \) in terms of the above parameters. Note that \( C \), \( A \) and \( L \) are the workload–independent cache geometry parameters; \( r \) is a cache–independent workload parameter; \( m \) depends on both the workload and the cache parameters.

One way to bound the average value of \( W \) is to compute the average number of instructions needed to refill an empty cache. (Recall that each processor, running the cache simulator with its assigned trace chunk, will start with an empty simulated cache). The number of cache lines is \( C/L \). First, assume a fully associative cache (\( A = C/L, c = 1 \)). In the extreme case, where every instruction is a load or store (\( r = 1 \)) and consecutive references loop through the \( C/L \) distinct lines, it would take \( C/L \) instructions to fill the cache. For \( r < 1 \), this (average) number would be \((C/L)/r\). Assuming that during the “fill” period for a given trace chunk, cache hits to the “filled” part take place at the same rate (= 1–m) as during the “steady–state” full trace processing (this is overly conservative), the overall bound on the overlap size reduces to:

\[
W = \frac{C/L}{(m \cdot r)} \quad \text{.......................... (2.1)}
\]

Thus, for example, if \( C = 64K, L = 128 \text{ bytes} \), \( m = 0.05 \) (5 \%) and \( r = 0.25 \) (25 \%), \( W = 40K \) instructions.

Generalizing to an \( A \)–way set associative cache does not change the above expression for \( W \), provided the accesses are uniformly distributed over all the congruence classes. The reader is referred to [10] for further discussion on the choice of \( W \). In section 3, we present experimental results to show the observed values of \( W \) which were adequate for minimizing the cold–start error. In general, we have found equation (2.1) to be a conservative, yet practical guide for choosing \( W \).
2.2 Trace sampling and parallel processing

Previous work on the cost benefits of trace sampling (e.g., [2,3] for cache/memory modeling and [4, 5] for general processor–cache cycle simulators) have shown the promise of this approach. Availability of a large–scale parallel processor like the SP–2 enabled us to arrive at appropriate sampling strategies for large commercial traces and the SPEC95 suite through extensive experimentation. Selected results are discussed in section 3.

2.3 Speed–up expectation

As discussed in detail in the full technical report [10], under (validated) assumptions of disk i/o behavior on the SP–2, the speed–up expectation for the parallel trace–driven simulation problem is linear for most practical tools in use. For a simulation tool whose native processing speed is much faster than the trace read/decode process, the speed–up expectation is sub–linear [10].

3. Experimental results and discussion

We restrict our data to three workloads for brevity: (a) gcc from the SPEC95 suite; (b) tomcatv from the SPEC95 suite and (c) a commercial TPC–C (henceforth referred to as: tpcc) workload trace. These traces exhibit a range of characteristics (e.g. hard–to–predict, branch intensive codes and loop intensive structured memory access codes) which are known to be difficult candidates for representative sampling methods.

3.1 Zero–state tools

First, we summarize the observed results for zero–state tools. We used a tool called trance+ (a trace analysis program) in a mode in which it outputs the instruction frequency mix of the workload, and also the idealized instruction issue width profile for a 3–issue super scalar machine. The errors (compared to a full trace, single processor run) were essentially zero, as expected, for all statistics produced by this tool. The reader is referred to [10] for the actual data.

The speedup characteristics for trance+ were found to be sub–linear. In this case, with the core speed of the tool being quite high (∼35,000 instructions per second), the overall runs were i/o bound, with the trace “filtration” (read/decode) process dominating.

In combining (uniform) sampling with parallel processing, one needs to use caution in selecting a sampling strategy (even for simple zero–state tools), especially for workloads with periodic behavior. This is discussed in detail in [10].

3.4 Finite–state tools

In this subsection, we report results based on finite–state tools. Most of the results reported are for H–timer, (a cycle–by–cycle processor timer) which was used in finite cache mode, with a 64K 4–way set associative level–1 data cache, a 32K 8–way set associative level–1 instruction cache and with infinite L2 cache. The processor modeled is a 4–issue super scalar processor, with two fixed point units (each capable of executing load–store and integer instructions), one floating point unit and a branch unit. The processor supports 16 outstanding instructions (corresponding to a reorder buffer of size 16), 2–bit branch prediction, speculative processing past up to 2 outstanding branches. This timer tool (with parametric variations) was used in an actual processor development project within IBM.

Figure 2 shows the speedup results (H–timer runs) for the chosen benchmarks using simple parallel trace chunking, without warm–start correction (overlapped chunks). Even with overlapped chunks, since the overlap sizes are small, the basic speed–up characteristics remain unchanged; hence they are not shown separately. The speed–up behavior is linear, as expected, with very slight deviation from the ideal slope.

Figure 3 shows the error in CPI (compared to full trace, single processor runs) against the number of processors (or chunks). No warm–start correction was used, because of the relatively large chunk sizes. The percentage errors are very small for up to 32 processors. In general, increasing the number of chunks causes each chunk size to decrease; eventually, beyond a certain number of chunks, this would exacerbate the cold–start (primarily cache) error. Incidentally, in addition to CPI, all other timer output statistics, (e.g. various queue/buffer usage histograms, cache miss rates, workload characterization metrics like instruction or basic block frequency mix, etc.) remain bounded in error by about ± 0.3 %. For example, Figure 4 shows the 16–entry reorder (completion) buffer utilization histogram, for n = 1 (single chunk, full trace), and n = 32, with the tpcc trace. Essentially, there is an exact match of the statistics.

For smaller full trace lengths, the errors grow more significantly as the number of parallel chunks is increased. In such cases, use of parallel overlapped chunks enables us to reduce the error margins back to almost zero. Table 1 shows this effect for runs made using a cache simulator (ksim). The data was found to be consistent with our earlier expectations (equation 2.1): it was validated by considering the workload miss ratios and the load/store frequencies.
Figures 5 through 7 show the variation in CPI error
vs simulation cost in various parallel sampling ex-
periments. For a given number of processors, n, the trace is
divided into n equal chunks and a sample is taken from
each chunk. For a given x–value (trace fraction), say 3.13
in Figure 5, each sample of the n=8 case contains
376,832 instructions, while each sample of the n=64 case
contains 47,104 instructions. Increasing x–values essen-
tially implies increasing the width of each sample.

Figure 5 shows the error characteristics for tpcc in the
absence of warm–start correction. Figure 6 shows this in
the presence of warm–start correction (W = 5K instruc-
tions). We see that for small values of total trace size
(simulation cost), the error is always quite high. This is
because the individual trace samples are relatively small, causing large cold–start cache miss ratio errors.

Let the overall average CPI value for a given work-
load be CPI real. In sampled trace simulation, we estimate
the real CPI as: CPI real ≈ (1/n) * Σ (CPIj) , where the
summation is taken over the n samples j = 1, 2, 3, ..., n and
CPIj is the measured CPI for the jth sample. There are
two main sources of error in such estimation: (a) error in
computing the true value of CPIj because of incorrect
initial state (e.g. cold cache error); and (b) statistical
sampling error.

The qualitative effect of (a) is clear: it adds a positive
effect to CPIj, and hence in the estimate of CPI real due to
an over–estimation of the cache miss ratio for the jth sample. Factor (b) can have a much more ill–defined effect
on the overall estimate. A selected sample may be
grossly unrepresentative (i.e. it may have a large devi-
ation from the actual population mean) in multiple cate-
gories (e.g. instruction frequency mix or branch
predictability) and yet it may produce a CPI value which
is close to CPI real. This is because CPI is an aggregate
metric, and there are often cancelling effects. Thus, the
sampling error characteristics for overall CPI are often
anomalous and hard to interpret.

For tpcc, it is clear from the data shown (Figures 5 and
6) that (a) above is by far the more dominating factor. For
lower values of simulation cost (e.g. 0.39 or 0.78), the
effect of cold cache error is clear. We see sharp decreases
in the error in this case, as we introduce warm–start correc-
tion (e.g. Figure 5 versus Figure 6). As the sample
width is increased such that more than 3 % of the trace is
consumed, the cold cache error diminishes. The number
of samples (n) becomes less important because there is
very little variation in the average CPI profile of tpcc. (In
other words, the statistical variance around the mean CPI
value of tpcc is quite low). This is not true for tomcatv,
which has distinct “loopy” regions. Hence n remains an
important factor, even for larger simulation costs in the
case of tomcatv (Figures 7a and 7b).

A fundamental principle which guides the sampling
error characteristics is the central limit theorem [9]. A
consequence of this theorem is that if we are sampling
from a population with unknown distribution, the sam-
pling distribution of the observed mean will be approxi-
mately normal provided the number of samples, n is
large. The normal approximation will generally be good
if n ≥ 30 regardless of the shape of the population.
However, in our case, because of cancellation effects, the
CPI error is not always minimized by simply making n ≥ 30.
This is evident from Figure 6, where (on statistical
grounds) we may have expected the error to be always
smallest for n = 32; however in reality, n = 16 is often
better.

For tomcatv (Figure 7), the CPI error behavior is even
more anomalous as there is a marked difference when
samples are taken after skipping the first 3 million
instructions in the trace (Figure 7a).
The reason is that tomcatv has several distinct phases in addition to repetitive (loopy) behavior. The error does not necessarily go down as the sample width is increased, or as more samples are chosen. In situations such as this, it is extremely risky to choose a “statistically plausible” sampling strategy. The preferred methodology for periodic benchmarks like tomcatv is to use a basic-block driven analysis and reduced trace generation approach (e.g. the SMART methodology [6]) and then to use simple multi-chunking (with overlap) as needed, for parallel speedup.

Detailed interpretation of the graphs presented (Figures 5 through 7) requires analysis of errors in estimating the individual components of the aggregate CPI value, such as instruction frequency mix, branch prediction ratio, various buffer utilization histograms, etc. Such analysis is ongoing and will be reported in later reports.

Figure 4. Reorder buffer utilization histograms

Figure 5. Error characteristics, parallel sampling

Figure 6. Error characteristics, parallel sampling (tpcc) with warm–start correction

Figure 7. Error characteristics, parallel sampling (tomcatv), no warm–start correction
REFERENCES


<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Icache Miss Ratios (%)</th>
<th>Dcache Miss Ratios (%)</th>
<th>No. of loads per instruction (%)</th>
<th>No. of stores per instruction (%)</th>
<th>Icache Miss Ratio Errors (%) for various warm–start correction values (W instructions)</th>
<th>Dcache Miss Ratio Errors (%) for various warm–start correction values (W instructions)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>W = 0</td>
<td>W = 50K</td>
<td>W = 100K</td>
<td></td>
<td>W = 0</td>
<td>W = 50K</td>
</tr>
<tr>
<td>gcc</td>
<td>0.81</td>
<td>1.00</td>
<td>23.9</td>
<td>9.4</td>
<td>2.05</td>
<td>1.24</td>
</tr>
<tr>
<td>tomctv</td>
<td>0.04</td>
<td>8.62</td>
<td>29.7</td>
<td>14.3</td>
<td>12.3</td>
<td>1.36</td>
</tr>
<tr>
<td>tpcc</td>
<td>3.59</td>
<td>4.69</td>
<td>22.1</td>
<td>10.1</td>
<td>0.54</td>
<td>0.18</td>
</tr>
</tbody>
</table>