Title
Reducing time and space costs of memory tracing

Permalink
https://escholarship.org/uc/item/8m33r7wj

Author
Gao, Xiaofeng

Publication Date
2006

Peer reviewed|Thesis/dissertation
Reducing Time and Space Costs of Memory Tracing

A dissertation submitted in partial satisfaction of the requirements for the degree Doctor of Philosophy in
Computer Science

by

Xiaofeng Gao

Committee in charge:
Professor Allan Snavely, Co-chair
Professor Larry Carter, Co-chair
Professor Scott B. Baden
Professor Richard Carson
Professor Jeanne Ferrante
Professor Richard Wolski

2006
Copyright
Xiaofeng Gao, 2006
All rights reserved.
The dissertation of Xiaofeng Gao is approved, and it is acceptable in quality and form for publication on microfilm:

University of California, San Diego

2006
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signature Page</td>
<td>iii</td>
</tr>
<tr>
<td>Table of Contents</td>
<td>iv</td>
</tr>
<tr>
<td>List of Figures</td>
<td>vii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>viii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>ix</td>
</tr>
<tr>
<td>Vita, Publications, and Fields of Study</td>
<td>x</td>
</tr>
<tr>
<td>Abstract</td>
<td>xii</td>
</tr>
<tr>
<td>1  Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1. The Importance Of Tools For Dynamic Memory Address Tracing</td>
<td>1</td>
</tr>
<tr>
<td>1. Memory Subsystem Performance Dominates Today’s Machines And Applications</td>
<td>1</td>
</tr>
<tr>
<td>2. Memory Performance Modeling Requires Simulation</td>
<td>3</td>
</tr>
<tr>
<td>3. Trace-driven Simulation Demands Memory Tracing</td>
<td>5</td>
</tr>
<tr>
<td>4. Other Major Usages of Memory Tracing</td>
<td>6</td>
</tr>
<tr>
<td>2. Challenges in Memory Tracing</td>
<td>7</td>
</tr>
<tr>
<td>1. Challenges in Space</td>
<td>7</td>
</tr>
<tr>
<td>2. Challenges in Time</td>
<td>8</td>
</tr>
<tr>
<td>3. Challenges in Availability</td>
<td>10</td>
</tr>
<tr>
<td>3. Thesis Statement</td>
<td>11</td>
</tr>
<tr>
<td>4. Structure and Contributions of this Thesis</td>
<td>11</td>
</tr>
<tr>
<td>1. Techniques to Reduce Time Overhead of Memory Tracing</td>
<td>11</td>
</tr>
<tr>
<td>2. An Asynchronous Tracing Scheme</td>
<td>12</td>
</tr>
<tr>
<td>3. Path Grammar Guided Trace Compression</td>
<td>12</td>
</tr>
<tr>
<td>4. Structured Path Histories</td>
<td>13</td>
</tr>
<tr>
<td>5. Trace Approximation</td>
<td>13</td>
</tr>
<tr>
<td>6. Study of Cross-platform Memory Tracing</td>
<td>14</td>
</tr>
<tr>
<td>2 Reducing Time Cost of Memory Tracing with Traditional Instrumentation Tools</td>
<td>15</td>
</tr>
<tr>
<td>1. Introduction to Metasim</td>
<td>16</td>
</tr>
<tr>
<td>1. Comparing Binary Instrumentation APIs</td>
<td>17</td>
</tr>
<tr>
<td>2. Cost of Binary Instrumentation</td>
<td>19</td>
</tr>
<tr>
<td>1. Measuring Overhead: Control Flow Interruption</td>
<td>20</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>3. Reducing Overhead Caused by Saving Machine States</td>
<td>24</td>
</tr>
<tr>
<td>4. Reducing Instrumentation Points</td>
<td>25</td>
</tr>
<tr>
<td>1. Chaining Memory Instructions</td>
<td>26</td>
</tr>
<tr>
<td>2. Delayed Instrumentation with Symbolic Execution</td>
<td>28</td>
</tr>
<tr>
<td>5. Regenerate Addresses from Recorded Register Values</td>
<td>33</td>
</tr>
<tr>
<td>6. Related Issues</td>
<td>35</td>
</tr>
<tr>
<td>7. Results</td>
<td>36</td>
</tr>
<tr>
<td>8. Limitations</td>
<td>39</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3. A Lightweight Asynchronous Tracing Scheme</td>
<td>41</td>
</tr>
<tr>
<td>1. Limitations of Synchronous Tools</td>
<td>42</td>
</tr>
<tr>
<td>2. An Asynchronous Instrumentation Scheme</td>
<td>45</td>
</tr>
<tr>
<td>3. Implementation and Optimizations</td>
<td>47</td>
</tr>
<tr>
<td>1. User’s Responsibilities</td>
<td>51</td>
</tr>
<tr>
<td>2. An Example of an Instrumented Block</td>
<td>52</td>
</tr>
<tr>
<td>4. Experiments and Results</td>
<td>53</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4. Reducing Space Cost of Memory Tracing</td>
<td>57</td>
</tr>
<tr>
<td>1. Path Grammar Guided Trace Compression</td>
<td>57</td>
</tr>
<tr>
<td>1. Motivation</td>
<td>57</td>
</tr>
<tr>
<td>2. Limitations of Current Compression Schemes</td>
<td>60</td>
</tr>
<tr>
<td>3. Creating Path Grammar from Static Analysis</td>
<td>62</td>
</tr>
<tr>
<td>4. PGGTC Online Procession</td>
<td>64</td>
</tr>
<tr>
<td>5. Results</td>
<td>67</td>
</tr>
<tr>
<td>6. Trade-off between Time and Space</td>
<td>68</td>
</tr>
<tr>
<td>2. Advanced Features of Structured Path Histories</td>
<td>69</td>
</tr>
<tr>
<td>1. SPH Alternation and Synthesis</td>
<td>72</td>
</tr>
<tr>
<td>3. Path Grammar Guided Trace Approximation</td>
<td>73</td>
</tr>
<tr>
<td>1. Static Data Flow Analysis</td>
<td>74</td>
</tr>
<tr>
<td>2. Selective Dumping</td>
<td>75</td>
</tr>
<tr>
<td>3. Generating Effective Addresses from Memory Signatures</td>
<td>76</td>
</tr>
<tr>
<td>4. Results</td>
<td>78</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5. A Study of Cross-platform Memory Trace-driven simulation</td>
<td>82</td>
</tr>
<tr>
<td>1. Snively Conjecture</td>
<td>83</td>
</tr>
<tr>
<td>2. Use Metasim Tracer to Detect Access Patterns</td>
<td>84</td>
</tr>
<tr>
<td>3. Experimental Setup</td>
<td>86</td>
</tr>
<tr>
<td>1. Platform Specific Issues</td>
<td>88</td>
</tr>
<tr>
<td>4. Comparison of Memory Access Patterns</td>
<td>90</td>
</tr>
<tr>
<td>1. Cache Simulation Results Comparison</td>
<td>91</td>
</tr>
<tr>
<td>2. Comparison of Loop Level Random Ratios</td>
<td>101</td>
</tr>
<tr>
<td>3. Dynamic Memory Reference Comparison</td>
<td>102</td>
</tr>
<tr>
<td>5. A Consideration of the Differences in Memory Access Patterns</td>
<td>102</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

1.1 von Neumann Architecture .................................................. 2
1.2 Typical Memory Hierarchy .................................................... 3

2.1 Reducing Instrumentation Points with Static Analysis .................. 27
2.2 Algorithm for Chaining Memory Instructions ............................... 28
2.3 Register Aliasing Supported in Delayed Analysis ........................ 31
2.4 Simplified Algorithm of Delayed Instrumentation ........................ 32
2.5 Instrumentation Log ........................................................... 34

3.1 Example: a Basic Block with Three Interesting Events .................. 43
3.2 Instrumented Code Segment using Synchronous Tools ..................... 44
3.3 Instrumented Block with an Asynchronous Scheme ........................ 46
3.4 The Most Visited Basic Block of CG.S Instrumented Using ALITER 56

4.1 A Control Flow Example ....................................................... 60
4.2 Time and Space Costs of Compressing Control Flow Traces ............... 69
4.3 Alternating Control Flow ...................................................... 71

5.1 VLIW Instructions each made up of three operations ..................... 89
5.2 Overall L1 Hit Rates from Alpha, Power4 and Itanium Traces .......... 92
5.3 Difference of Loop Level Cache Hit Rates ................................ 93
5.4 Difference of Loop Level Cache Hit Rates ................................ 94
5.5 Difference of Loop Level Cache Hit Rates ................................ 95
5.6 Difference of Loop Level Cache Hit Rates ................................ 96
5.7 Difference of Loop Level Cache Hit Rates ................................ 97
5.8 Difference of Loop Level Cache Hit Rates ................................ 98
5.9 Difference of Loop Level Cache Hit Rates ................................ 99
5.10 Difference of Loop Level Cache Hit Rates ................................. 100
5.11 Random Ratio: Alpha vs. Itanium .......................................... 103
5.12 Random Ratio: Alpha vs. Power4 .......................................... 103
5.13 Difference in Dynamic Memory Reference Counts: Alpha vs. Itanium 104
5.14 Difference in Dynamic Memory Reference Counts: Alpha vs. Power4 105

6.1 PMaC Performance Prediction Framework .................................. 109
6.2 Static Binary Rewriter Model ............................................... 115
6.3 Dynamic Binary Patching Model ............................................ 116
6.4 JIT Instrumentation Model .................................................... 117
# List of Tables

2.1 Sample Slowdowns of Different Tools: ........................................ 17
2.2 System Setup .................................................................................. 23
2.3 Instrumentation Overhead using ATOM: .......................................... 24
2.4 Instrumentation Overhead using PIN: .............................................. 24
2.5 Instrumentation Overhead using DyninstAPI: .................................... 24
2.6 Instructions Supported by Symbolic Execution .................................... 30
2.7 Reduction in Number of Instrumentation Points ................................. 37
2.8 Comparison of Instrumentation Overhead: Experiment 0 ................. 38
2.9 Comparison of Instrumentation Overhead: Experiment 1 ................. 38
2.10 Tracing Time Comparison in seconds (slowdown in parenthesis) .......... 39

3.1 Slowdown of Acquiring Addresses Using Different Tools .................... 54
3.2 Slowdown of Acquiring Addresses .................................................. 54

4.1 Compression Ratios and Compressing Time: Part 1 .......................... 68
4.2 Compression Ratios and Compressing Time: Part 2 .......................... 68
4.3 Total Number of Instrument Points After and Before Static Analysis .... 74
4.4 Size of Full trace, Selective Dumped Trace and Memory Signature .... 79
4.5 Size of Memory Signature and SyncInfo ........................................ 79
4.6 Average Absolute Errors of Using Approximate Traces ..................... 80
4.7 Average Absolute Errors of Using Approximate Traces ..................... 80

5.1 System Configurations ..................................................................... 87
5.2 Slowdowns of Different Tools ......................................................... 88
5.3 Cache Configurations Simulated and Reported .................................. 91

6.1 Cache Configurations Simulated: part 1 ........................................ 111
6.2 Cache Configurations Simulated: Part 2 ...................................... 112
I am deeply appreciative of the aid given me, by my advisor and friend, Allan Snavely. It is most fortunate that I found him as my advisor. His unparalleled guidance and encouragement supported me through to the end of my program. His selflessness and willingness to help have made all my work, including this thesis, possible.

Furthermore, my thanks are due to Larry Carter and Jeanne Ferrante, who were both professors and mentors of mine, for all their insight and support.

My fellow PMaCers and lab-mates: Wayne, Laura, Barbara, Nicole, Mike, Mikey, Jihua, Cynthia, Jon, and especially Beth were such a pleasure to work with. I want to thank also, a friend who prefers the name Neyney, who helped in my adjustment to living in the USA.

Finally I want to thank my parents. Whatever I have accomplished, they deserve to share in the credit for that achievement.

Chapter 2, in part, is a reprint of the material as it appears in “Reducing Overheads for Acquiring Dynamic Memory Traces”, by Xiaofeng Gao, Michael Laurenzano, Beth Simon and Allan Snavely, published in the proceedings of 2005 IEEE International Symposium on Workload Characterization (IISWC05) Oct.05 Austin [28]. Chapter 3, in part, is a reprint of the material as it appears in “ALITER: An Asynchronous Lightweight Instrumentation Tool for Event Recording”, by Xiaofeng Gao, Beth Simon and Allan Snavely, published in the proceedings of the 2005 Workshop on Binary Instrumentation and Applications (WBIA05) Sept.05 St. Louis [29]. The dissertation author was the primary researcher and author; the co-authors listed on these publications directed, supervised, or helped the research which forms the basis for these chapters.
VITA

1975  Born
1997  B.S. Peking University
2000  M.S. Peking University
2000–2001  Teaching Assistant, University of California, San Diego
2001–2006  Research Assistant, University of California, San Diego
Summer 2003  Intern, Microsoft Research
Summer 2004  Intern, Microsoft Research
2006  Doctor of Philosophy
        University of California, San Diego

PUBLICATIONS

“Path Grammar Guided Trace Compression and Trace Approximation” Xiaofeng Gao, Allan Snavely and Larry Carter, Submitted for publication


“ALITER: An Asynchronous Lightweight Instrumentation Tool for Event Recording” Xiaofeng Gao, Beth Simon and Allan Snavely. Workshop on Binary Instrumentation and Applications (WBIA05) Sept.05 St. Louis

“Reducing Overheads for Acquiring Dynamic Traces” Xiaofeng Gao, Michael Laurenzano, Beth Simon and Allan Snavely. IEEE International Symposium on Workload Characterization (IISWC05) Oct.05 Austin


“Performance Modeling of HPC Applications” Allan Snavely, Xiaofeng Gao, Cynthia Lee, Nicole Wolter, Jesus Labarta, Judit Gimenez and Philip Jones. ParCo, Oct.03 Dresden


FIELDS OF STUDY

Major Field: Computer Science
   Computability and Complexity.
   Processor Li’ang Zhang

Major Field: Computer Science
   Performance Modeling and Prediction.
   Professors Allan Snavely and Larry Carter
ABSTRACT OF THE DISSERTATION

Reducing Time and Space Costs of Memory Tracing

by

Xiaofeng Gao

Doctor of Philosophy in Computer Science
University of California, San Diego, 2006
Professor Allan Snavely, Co-chair
Professor Larry Carter, Co-chair

Event tracing of applications under dynamic execution is crucial for performance modeling, optimization and trace-driven simulations. However, collecting and processing events, especially memory addresses, is extremely expensive in terms of time and space requirements. It is also challenging to find the right platform and the right tools to perform tracing. Such challenges greatly hinder the feasibility of tracing memory of large, long running, parallel applications.

In this thesis, the challenges in tracing memory are explored and several solutions are exhibited to face each challenge. The philosophy of these solutions, schemes and workarounds is to find balance in the time and space on available platforms with available tools.

Specifically, the time required to acquire memory traces can be greatly reduced by carefully identifying all causes of slowdown and addressing them in the design of built-for-the-purpose tracers. Techniques, including buffering, chaining and delayed instrumentation, are introduced and have been shown to reduce the time cost of memory tracing by more than 80% when used with traditional instrumentation tools. In addition, a lightweight instrumentation tool ALITER, which only causes two-fold slowdown in collecting full memory traces, is introduced to demonstrate the benefits of asynchronous tracing schemes.
Path grammar guided trace compression and trace approximation are explored in this thesis to reduce space costs of memory tracing. The efficacy of low-level general purpose compression schemes is greatly enhanced when they are organized around information about program structure and phases; combined with trace recording designed to capture the locality properties of random events, not the exact random events themselves, space required to store traces can be reduced by many orders of magnitude. These techniques enable one to generate reusable intermediate representations of memory traces, which are small enough to be stored on the disk, accurate enough for trace-driven simulations and fast enough to collect and process.

Together with working through practical issues of availability, these advances make trace-driven simulation of full scale HPC applications feasible.
Chapter 1

Introduction

1.1 The Importance Of Tools For Dynamic Memory Address Tracing

1.1.1 Memory Subsystem Performance Dominates Today’s Machines And Applications

Computer vendors often tout metrics pertaining to the CPU (central processing unit) such as clock frequency or theoretical peak FLOPS (floating point operations per second) to advertise their machines. However, sophisticated users have long been aware that the actual performance of an application, in terms of realizable operations per second, is likely to be dominated by the time taken by operations that interact with the memory subsystem. This is because most current computers are based on the von Neumann computation model [92] shown in Figure 1.1. The model requires data and instructions first be fetched into the processor from the memory before they can be processed, and the results often stored back to memory. Unfortunately, the speed of memory technology has not kept up with CPU speed and the gap is steadily increasing. Such discrepancy in speed leads to what is termed the von Neumann bottleneck [8], indicating that no matter how fast the CPU can execute instructions, overall performance is most likely limited by how fast data can be delivered into the processor. This is partic-
ularly true of HPC (high performance computing) applications, which often deal with large amounts of data.

![Figure 1.1: von Neumann Architecture](image)

The separation of storage from the processing unit is implicit in the von Neumann architecture.

From the above, it should be clear that understanding the requirements an application places on the memory subsystem, and the raw capabilities of the memory subsystem, are key to understanding, and potentially improving, the performance of modern applications and computers. Performance modeling is the science of understanding and improving computer performance; creating accurate memory performance models is then a key component of performance modeling overall. In fact, research indicates that by only modeling memory performance, one can explain 80% of performance differences between most modern HPC architectures. On the other hand, models that do not consider memory performance carefully can yield errors over 60% [18, 54]!
1.1.2 Memory Performance Modeling Requires Simulation

Particularly in the last 15 years, caches have become popular to alleviate some of the constriction of the von Neumann bottleneck. Caches are smaller, but faster and more expensive, memories that are placed between main memory and the CPU [65]; they can be used to hide the speed gap between the CPU and DRAM (dynamic random access memory). If the most frequently used and the most likely to be used data can be stored in these faster memories, they can be delivered to the CPU faster. Figure 1.2 shows a typical multiple layered memory system hierarchy [4]. Caches can be very effective because many applications tend to have high locality properties in their memory access patterns. Locality in memory access patterns means if one address is accessed, it is very likely to be accessed again in the near future. It also means the close neighbors of the accessed addresses are very likely to be accessed. These two properties are referred to as temporal locality and spatial locality respectively. Although caches are extremely useful in improving performance, they dramatically increase the complexity of the computer and thus also the complexity required for accurate performance modeling.

![Figure 1.2: Typical Memory Hierarchy](image)

The linear model [71, 72] shown in Formula 1.1 is a simple but illustrative
memory model useful for memory performance modeling. This model assumes all cache misses in the same level of cache cause the same amount of penalty in time. Practically speaking, such penalties can either be measured using micro-benchmarks \[81, 36\], or estimated from vendors’ specifications.

\[
\text{ExecutionTime} = \frac{\text{TotalWorkload}}{\text{PeakExecutionRate}} + \sum_{k=1}^{\text{cachelevel}} \text{TotalMiss}_k \times \text{MissPenalty}_k.
\]

To apply a simple a linear model like this to the task of estimating execution time of an application, a key sub-task would be to determine the cache miss rates at each level of the memory hierarchy for each phase of the application’s execution. This unfortunately can be quite difficult in general. The miss rates depend on how much locality each phase of the application naturally has and also depends on some machine attributes such as how well the cache configurations (size, and replacement algorithm etc.) suit the memory access patterns of the application. Neither of these two can be easily quantified independently.

Cache hit rates can often be observed for the entire program by consulting hardware performance counters. Performance counters are a set of special hardware registers one can use to monitor certain events, such as cache misses, page fault, etc. Although such counters are very inexpensive to use, the results are too coarse for accurate performance modeling as they come from the hardware level and are uncorrelated to the software level. Once the counters are started, all events from every phase, and even events from other processes including the operating system, will be counted and lumped together. In addition, these counters are NOT typically guaranteed to be exact, accurate or even correct (for example performance research scientists at IBM distrust IBM’s performance counters). Most fundamentally as a limitation, the results of hardware counters on one machine are not generalizable in any obvious way to other current or future machines.

Many methods have been proposed to predict cache miss rates for ar-
arbitrary cache configurations cheaply from static analysis of applications’ source codes or binaries. These are worthy attempts to make cache miss rate estimation machine independent. However, static analysis is not likely to work for all the cases. An indirect memory reference which is input dependent will break all such analysis because the access pattern is not known until runtime.

Given there is no easy way even to get miss rates right, much research in performance modeling resorts to full-scale trace-driven simulation. Simulators can be written to closely mimic what actually happens when a processor executes a given instruction mix, thus generating accurate prediction if they are used correctly. Typically one adds some instrumentation to an application of interest such that, when the application is run, a record is collected of the instructions it issued (including memory operations); this recorded trace can then be replayed through the simulator either to study directly the behaviors of the application in each phase or to model the performance impact of proposed architectural innovations different from the machine where the trace was recorded. Today 80% to 90% of research related to architectural design relies on trace-driven simulation for validation. The percentage in fact has shown steadily increasing over the past several years [79].

1.1.3 Trace-driven Simulation Demands Memory Tracing

Most work pertaining to tools for trace-driven simulation focuses on details of the simulator; how to model the internal mechanism of the hardware with high-definition. This work focuses on the traces themselves. Any trace-driven simulation will need to first get the traces. In fact, processing events closely following hardware specifications is only one aspect of a successful simulation framework. Without valid input data of sufficient size and scope, simulations may be of limited relevance to the performance of proposed computers on full-scale applications. How to collect, represent and store representative traces is then a fundamental research problem of wide ranging impact. An accurate simulation demands not

\[1\] It does not mean simulations will always generate accurate results [32]
just any trace, but traces that are complete and representative of applications’ requirements. Recently, collecting complete and representative traces has been so extremely expensive in time and also in space required as to make it impractical to record them to disk. Worse, for many platforms, it is not even possible to collect traces because the enabling hardware or software support does not exist. The challenges are the focus of the thesis and are further explained in Section 1.2.

1.1.4 Other Major Usages of Memory Tracing

Memory traces are not only used for performance modeling and prediction via simulation. They have been widely used for code optimization and debugging purposes. Much work has been done, for example, in reorganizing data layout based on profiled memory access data. Such optimizations [21, 43] can significantly increase cache hit rates, or enable more efficient prefetching, thus improving applications’ overall performances. However, most of these approaches require multiple steps. The program is first profiled to collect data and then it is optimized based on the profiled data. One major pitfall of such off-line profiling techniques is that the profiled data may not be characteristic of future runs. Thus selected optimizations based on the profiled data may not be helpful and could even be harmful. On-the-fly profiling is extremely desirable, but it is rarely implemented simply because of the dreadful slowness of memory tracing (something improved upon by this thesis.)

Memory instrumentation and tracing is also widely used in debugging tools. Traditionally, memory tracing is mostly used to detect invalid accesses to a particular region in the memory. Nowadays more and more debugging tools use memory tracing in shared memory, multiple threads environment to detect race conditions, or synchronization errors. The usefulness of such tools largely depends on their speed and responsiveness which, historically, has been dismal.

The work described in this thesis originated from trace-driven simulation used for performance modeling and prediction. However the challenges of tracing
memory, no matter for what purpose, are universal. The solutions developed to reduce time and space overheads of memory tracing, described in this thesis, are applicable the several memory tracing purposes described above.

1.2 Challenges in Memory Tracing

1.2.1 Challenges in Space

The most common way to collect event traces is to instrument the program’s source code or binary and then execute the instrumented version of the program, thereby to capture and record events and associated information. The recorded events and associated information are then stored into a file. In practice, event traces can be enormous. Current processors can steadily issue more than 100 million memory instructions per second. Assuming each address takes 4 bytes to represent, it would require several terabytes of disk space to record the memory addresses issued by a single processor program that runs for an hour. The disk space required to save complete memory traces for parallel applications running on multiple processors is beyond current capacity of on-line disk systems; average users will run out disk space for acquiring memory traces for full-scale applications. Thus, how to reduce the disk space required to store traces is one of the major challenges needing to be met for moving trace-driven simulation into the realm of the feasible for full-scale HPC applications.

Traces can be compressed to save disk space. However none of the available compression schemes are guaranteed to work well on memory traces. Many algorithms require scratch disk to save the full trace first before compression [63], such a requirement is not practical for memory traces as already observed; users will run out of disk space to save them in the first place. For other accumulative compression algorithms, based on the published results, compression ratios vary from over one million to as little as 1 [24, 22], and there is no way to estimate the ratios beforehand. Such uncertainty in the compression ratios is very undesirable
because for a previously untraced application, it might take several kilobytes to save the traces, or it might take several terabytes. So one would not know before starting if one had sufficient disk space or not.

The compression ratio achieved not only depends on how well compression algorithms can find repetitive patterns in the trace and utilize them for compression, it also depends on whether the trace is inherently compressible or not. Intuitively, a trace of random events with no relation to each other is incompressible; none of the available compression algorithm can achieve high compression ratios on these. In fact, the sizes of such traces are unlikely to be reduced dramatically with any lossless compression scheme.

Trace sizes can be reduced by discarding some none critical events in the sequences. Events are passed through a filter and if an event is not going to change the simulator state, it could be discarded from the trace. Such techniques generally are closely tied to high-level simulators, such as simulators simulating page table replacement policies. For exact simulation, such techniques discard too much data and make the simulator less accurate.

Filtering technique do not work in any case for traces whose contents are mostly random. Most of these random events are likely to change the simulator state thus a majority of these random events will be kept in the “reduced” trace. In fact, random events are the bane of trace compression. Compression algorithms constantly have trouble with non-patterned elements. Many rules and patterns are created to try to summarize event sequences that have no pattern at all. How to deal with events that appear random is a major task to solve in order to reduce application’s trace sizes satisfactorily. This issue is addressed in this thesis.

1.2.2 Challenges in Time

Alternatively, the disk space requirement can be eliminated by inserting simulators or analysis routines directly into the application’s binary. Events, such as effective addresses, are captured and consumed by simulators right away without
saving them. When the instrumented binary completes, only a compact report is generated (as for example cache miss rates predicted for each phase). Although the space requirement is eliminated, the instrumented binary runs extremely slowly. It is not uncommon to experience over 2000 times slowdown (slowdown measured as a multiple of the run time of the un-instrumented code). For a short program that completes in 10 minutes, the instrumented version will probably take more than 300 hours to complete!

For example, a recent performance modeling effort [19] on behalf of Department of Defense High Performance Computing Modernization Office (DOD HPCMO) used several million processor hours on supercomputers at Army Research Laboratory (ARL) [5], Arctic Region Supercomputing Center (ARSC) [7], US Army Engineer Research and Development Center (ERDC) [26], The Naval Oceanographic Office (NAVO) [60], the Pittsburgh Supercomputing Center (PSC) [68], and the San Diego Supercomputer Center (SDSC) [75] in 2004. Most of the time was spent running the instrumented versions of seven large strategic applications to gather statistics on, and simulate many current and future memory subsystems against, their memory accesses. The result was deep insight gained into several strategic applications and related supercomputer performance [18, 19]; however a wider community interested in memory performance may not even be able to gain access to this many supercomputer hours which roughly translates to a dedicated 500 processor system devoted only to tracing 24 hours a day 7 days a week for a year. The level of effort, never mind the huge amount of HPC resources consumed, is beyond the regime that most users could endure no matter how useful the resulting data might be.

In addition, because traces are processed on-the-fly and there is no actual trace saved, any modification in the analysis routines or simulators requires rerunning the slow instrumented version of each application. The user has to once again endure the tremendous slowdown in time, and pay for the extra overhead caused by instrumentation (as we will see in the later chapters, these could be major costs).
Reducing these onerous time overheads is a contribution of this thesis.

1.2.3 Challenges in Availability

Even without considering time and space, it can be quite a challenge to find a suitable platform to collect memory traces. When trace-driven simulations are used to evaluate the possible benefits of some innovation, people commonly use memory traces collected on other platforms to simulate these new systems designs. Such a cross-platform simulation approach is used because the hardware with those proposed features simply does not exist (it is the design under investigation via simulation).

Even if the hardware does exist, cross-platform tracing is still commonly used because the target platform lacks usable tools. For example currently there is no usable tool for collecting memory traces on AMD Opteron processors so it is simply not possible to collect memory traces on such a platform. Even if there are instrumentation tools, they may not be usable for one’s purpose. For example, there are several instrumentation tools available on IBM POWER processors. Some of them can even be programmed to collect memory traces. However they are not practically usable for tracing larger application due to their extremely high overhead.

It is not hard to understand that memory traces could be different on different platforms, especially when the innovation itself is related to the memory system. Thus the simulated results in such a cross-platform simulation scheme may not be correct. Unfortunately for expense reasons, actual hardware may never be created to incorporate a proposed design. This leaves cross-platform simulation as the only viable option to evaluate many research designs. A real challenge is to find the right platform to collect traces from so there will be verisimulitude in the simulated results. This issue is explored in this thesis.
1.3 Thesis Statement

The time required to acquire memory traces can be greatly reduced by carefully identifying all causes of slowdown and addressing them in the design of built-for-the-purpose tracers. The efficacy of low-level general purpose compression schemes is greatly enhanced when they are organized around information about program structure and phases; combined with trace recording designed to capture the locality properties of random events, not the exact random events themselves, space required to store traces can be greatly reduced. Together with working through practical issues of availability, these advances make trace-driven simulation of full-scale HPC applications feasible.

1.4 Structure and Contributions of this Thesis

In this thesis, the three major challenges of memory tracing: time, space and availability are discussed. Various solutions are introduced to tackle these challenges so memory tracing can be made practical and affordable for long running applications. The work presented in this thesis originated from a collaborative effort to build a performance modeling and prediction framework for large parallel applications. In Chapter 6.1 and Chapter 6.2, the PMaC performance modeling and prediction framework and its components are briefly introduced. Since many of the solutions in this thesis are affected by the nature of binary instrumentation tools, the basic principles of binary instrumentation and typical implementations are summarized in Chapter 6.3. Other related work is presented in Chapter 6.4.

The main body of this thesis is organized into 4 chapters covering six major contributions, covering various aspects related to memory tracing as follows.

1.4.1 Techniques to Reduce Time Overhead of Memory Tracing

Chapter 2, identifies the major causes of the high overheads normally experienced when using binary instrumentation tools and analysis on-the-fly. Tech-
niques of asynchronous processing are demonstrated to be extremely effective in reducing the costs of saving machine state and the performance degradation caused by cache interference. Also demonstrated is how to use static analysis, symbolic execution and delayed instrumentation to dramatically reduce the number of instrumentation points required for collecting full memory traces. With all these techniques implemented, the time overhead of memory tracing is reduced by 80%.

1.4.2 An Asynchronous Tracing Scheme

In Chapter 3, a new instrumentation scheme: ALITER(Asynchronous Lightweight Instrumentation for Trace Event Recording) is proposed and implemented. ALITER is extremely lightweight compared to generic synchronous binary instrumentation tools such as ATOM [85]. Asynchronous event recording eliminates frequent control transfers by first saving addresses in an ALITER maintained buffer and by calling user’s routines only when the buffer is full, thus overhead caused by saving machine state required at each control transfer back and forth from instrumented code is significantly reduced. ALITER also inlines aggressively the instructions for event recording. Inlining enables ALITER to reschedule instructions inserted for instrumentation together with the original ones, and merge common actions, thus the resulting binaries are much better performing on the targeted hardware. With such a lightweight instrumentation tool, one can enjoy only 2 fold slowdown for collecting full effective address traces, instead of the at least 10 fold slowdown experienced using other generic binary instrumentation tools.

1.4.3 Path Grammar Guided Trace Compression

In Chapter 4, a new trace compression scheme, path grammar guided trace compression (PGGTC) is introduced to address the challenge in space required to save traces. Unlike most current compression schemes which find rules for compression from dynamic trace sequences, PGGTC relies on static analysis of control flow graphs (CFG) to build the rules and grammars that will be used for
compression, thus eliminating much of the workload of creating, maintaining and looking up rules and actions that other algorithms suffer from. It has also eliminated the problem of ever increasing look-up table size that troubles some other dynamic compression schemes. PGGTC, when combined with gzip, is 40 times faster than the popular Sequitur [63] algorithm, and generates compressed traces only 3 times larger. Furthermore PGGTC with gzip does not require the full trace to be first stored on scratch disk as Sequitur does. PGGTC can be also used with Sequitur on existing traces. Such combination can double the compression ratios of Sequitur and speedup the process by 14 times.

1.4.4 Structured Path Histories

PGGTC compresses control flow traces into Structured Path Histories (SPH). These structured path histories are used as the backbone of execution for generating effective addresses. SPHs have many desirable features that can help in understanding applications’ behaviors. SPHs are easy to handle because they can be decompressed and analyzed individually. SPHs also maintain instance markers so one can easily locate the behaviors of a particular instance from its structured path history. SPHs can be used to find differences among traces of the same application with different inputs, so one can easily identify the stable structures from input-dependent ones. SPHs can also be artificially generated and plugged into histories that are actually traced. This way a simulator can easily cover all interesting scenarios systematically without actually collecting all the traces over and over again. These features are explained in detail in Chapter 4.2.

1.4.5 Trace Approximation

As mentioned above, random addresses are the bane of compression algorithms and none of the lossless compression algorithm can compress random traces with significant compression ratio. Chapter 4.3 introduces memory Trace Approximation (TA) to store address traces that contain many random elements
efficiently and effectively.

Trace approximation leverages the following idea: many times, for trace-driven simulation and validation of benefit of architectural features, one would be satisfied with a “performance similar” trace. That is, from the performance standpoint, it may be that any sequence of random memory accesses with the same statistical distribution would perform the same against the architectural feature being evaluated. Therefore one can replace traces of random addresses with other random addresses that are more compressable. Using TA technique one can obtain a compact event trace from a program that, when replayed against a simulator, will reproduce user-defined statistical behavior of the events of the original program without requiring exact duplication of particular events in sequence (such as the exact sequence of random memory addresses touched).

1.4.6 Study of Cross-platform Memory Tracing

Chapter 5 studies the memory access patterns of several selected applications on three different computer platforms based on the Alpha, Power4 and Itanium processors. The memory access patterns are compared at the loop level in terms of dynamic memory counts, random ratio and cache hits. The studies reveal many problems of a naive cross-platform memory tracing scheme. Dynamic memory counts for the same loop can be dramatically different due to architecture related issues and compiler related issues. On the other hand, simulated cache hit rates and raw statistics have relatively smaller differences. However the smaller differences do not necessarily mean these traces can be used interchangeably. This chapter highlights the possible problems with cross-platform memory trace-driven simulation: although memory traces of the same application on different platforms are largely similar, dramatic differences do show and cause problems with fine-grained analysis.

Following the main body of contributions, future directions for this work is discussed in Chapter 7. Finally Chapter 8 presents conclusions.
Chapter 2

Reducing Time Cost of Memory Tracing with Traditional Instrumentation Tools

Instrumented binaries execute painfully slowly. In fact the slowness dramatically limits broader usage of memory tracing for either performance modeling or for optimization. Table 2.1 lists the typical slowdown experienced in the DOD HPCMO T105 effort. During that effort, 7 strategic applications were selected to be instrumented and analyzed. Despite the deep insights gained in understanding these applications’ memory behaviors, more than 1 million CPU hours were consumed, which roughly translates to a dedicated 500 processor system devoted only to tracing 24 hours a day 7 days a week for a year. Such requirement is way beyond any general user’s reach, no matter how useful such information may turn out to be.

The data in Table 2.1 is from traces that include significant performance improvements from buffering techniques which will be explained in Section 2.3. Naively inserting cache simulators before each memory instruction will cause much higher slowdown than reported in Table 2.1. Just for a simple example to illustrate the slowness, if one naively instrumented CG.S and FT.S with Metasim tracer [82],
every memory instructions in the binaries will be instrumented and the captured addresses will be processed right away. The instrumented FT is 1368 times slower than the original one and the slowdown for CG is 2857 times! Such astonishing slowdown makes simulation on-the-fly nearly impractical for any real applications. If an application runs for an hour, the instrumented version will take months to complete.

In this chapter, the causes of the slowdown in binary instrumentation are first examined and then three techniques that are shown to be extremely effective in reducing the overhead caused by binary instrumentation are introduced.

2.1 Introduction to Metasim

Metasim tracer [82] is a tool used for collecting application signatures in the PMaC performance prediction framework [83]. Metasim tracer collects memory trace information and computes statistics on the address stream on-the-fly without ever storing the raw traces. It has been implemented on top of three different low-level instrumentation APIs, ATOM [85], Dyninst [12], and PIN [51], and has been deployed on several different HPC platforms including ones based on the Alpha, Power3, Power4, and Itanium 2 processors. It is available for download at www.sdsc.edu/pmac. Despite some differences in technical details of the instrumentation APIs, the underlying idea and the analysis routines are the same across the APIs and platforms. The more detailed description of the PMaC framework and the tool can be found in Chapter 6.1.

There are two forms of pre-defined probes used in Metasim tracer. One probe is inserted before each memory instruction to capture effective addresses and perform analysis, such as detecting memory access patterns and simulating several cache configurations. Additionally, another probe is inserted at the end of each basic block to capture information at the basic block level and to control the sampling state.
Table 2.1: Sample Slowdowns of Different Tools:

Results from TI05 Study, Slowdown is measured as the multiple of execution

time of the instrumented binary over the execution time of the original code.

Cells marked as −− indicates the experiment is too slow to complete in the given
queue limit.

<table>
<thead>
<tr>
<th>Application</th>
<th>ATOM</th>
<th>PIN</th>
<th>DyninstAPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG.A.4</td>
<td>98.82X</td>
<td>222.67X</td>
<td>896.86X</td>
</tr>
<tr>
<td>FT.A.4</td>
<td>44.22X</td>
<td>127.64X</td>
<td>1054.70X</td>
</tr>
<tr>
<td>MG.A.4</td>
<td>107.69X</td>
<td>168.61X</td>
<td>989.53X</td>
</tr>
<tr>
<td>LU.A.4</td>
<td>80.72X</td>
<td>153.46X</td>
<td>−−</td>
</tr>
<tr>
<td>SP.A.4</td>
<td>67.56X</td>
<td>93.04X</td>
<td>−−</td>
</tr>
<tr>
<td>CG.B.8</td>
<td>26.96X</td>
<td>163.05X</td>
<td>−−</td>
</tr>
<tr>
<td>FT.B.8</td>
<td>27.48X</td>
<td>88.18X</td>
<td>−−</td>
</tr>
<tr>
<td>MG.B.8</td>
<td>97.18X</td>
<td>161.01X</td>
<td>−−</td>
</tr>
<tr>
<td>CG.B.16</td>
<td>25.49X</td>
<td>131.22X</td>
<td>−−</td>
</tr>
<tr>
<td>FT.B.16</td>
<td>14.93X</td>
<td>99.50X</td>
<td>−−</td>
</tr>
<tr>
<td>MG.B.16</td>
<td>101.18X</td>
<td>107.75X</td>
<td>−−</td>
</tr>
<tr>
<td>CG.C.8</td>
<td>21.68X</td>
<td>225.06X</td>
<td>−−</td>
</tr>
<tr>
<td>FT.C.8</td>
<td>50.42X</td>
<td>96.87X</td>
<td>−−</td>
</tr>
<tr>
<td>MG.C.8</td>
<td>55.81X</td>
<td>112.52X</td>
<td>−−</td>
</tr>
<tr>
<td>CG.C.16</td>
<td>15.02X</td>
<td>107.77X</td>
<td>−−</td>
</tr>
<tr>
<td>FT.C.16</td>
<td>11.59X</td>
<td>84.87X</td>
<td>−−</td>
</tr>
<tr>
<td>MG.C.16</td>
<td>34.67X</td>
<td>126.61X</td>
<td>−−</td>
</tr>
</tbody>
</table>

In what follows, it is important to realize that the analysis code snippet
which gathers effective addresses will be visited much more frequently than the
snippet at the end of each basic block. On Alphas for example, the basic blocks
for major loops normally are unrolled 8 to 16 times. It is not uncommon to see more
than 100 memory instructions in one basic block. Any reduction in the overhead
caused by effective address monitoring will have a significant performance impact.
Overhead of basic block tallying is less critical.

2.1.1 Comparing Binary Instrumentation APIs

Three well known instrumentation APIs were used in this work. Each API
instruments the binary codes with a different process and has different costs. More
detailed background information about binary instrumentation tools can be found in Chapter 6.3. In Table 2.1 is shown a comparison of the overhead exhibited by each tool on the dynamic memory tracer from the HPCMO effort in 2005. One of the APIs, Dyninst, has such high overheads that even tracing benchmarks becomes prohibitive, so only a few representative results are run and reported.

ATOM [85] is a binary rewriting tool available on Alpha processors. It is the pioneer of binary instrumentation tool and the most used instrumentation tool by academic researches. As a static binary rewriter, ATOM creates a new binary from the original one guided by user defined instrumentation and analysis routines. ATOM has flexibility in placing analysis routines as a binary rewriter. It also tries to inline instrumentation snippets into the original code and performs certain performance optimizations if the snippets are simple. The users can force ATOM to perform (using -A1 flag) cross-routine analysis between the original instructions and instrumented ones so the new binary will run faster.

However instrumentation is done pre-execution so the user has to specify a priori all the places in the program where interesting events might happen. In addition, the analysis snippets are integrated into the new binary; there is no way to remove them during a run. These snippets will cause overhead even if the analysis portion of the snippets are occasionally bypassed (e.g. for sampled data collection).

PIN [51] is a Just-in-time (JIT) instrumentation API available for Intel processors (IA-32 and IA-64). PIN utilizes a code cache to instrument and keep the instrumented code sequence in memory. Within the code cache, PIN performs limited optimization to reduce overhead.

DyninstAPI [12] is a cross-platform API available on most common platforms including those supported by PIN and ATOM. It is based on the model of dynamic code patching. It has the all the flexibility of dynamic patching tools. The user can program instrumentation in such a way that snippets can be inserted and removed dynamically during the application’s runtime. Ideally with such ability,
sampling of data collection could be implemented very efficiently by running an instrumented binary when data collection is turned off and only paying the overhead of instrumentation when sampling is turned on. However this flexibility comes with very a high price tag. The snippets are installed by FAR JUMP instructions in the program that then point to a base trap. Then from the base trap, there is a JUMP to the entry of the corresponding snippet. Finally there is another jump to go back to the original binary. The cost of these three jumps is considerably heavier than the cost of calling the snippets with the other two APIs. In addition, DyninstAPI lacks cross routine analysis between the original instructions and the instrumented ones. It will save all general purpose registers registers when calling an instrumentation snippet, no matter whether they are used or not by the snippet. Such a simplistic implementation results in extremely high overhead as shown in Table 2.1. this overhead renders instruction level instrumentation with Dyninst impractical for all but small benchmarks.

2.2 Cost of Binary Instrumentation

Metasim tracer uses SimPoint [77] guided sampling to reduce the number of addresses required to be processed. On average with SimPoint, only about 1% of all dynamic addresses are needed to be processed while still very accurately representing the behavior of all dynamic addresses via interpolation [50].

Even after employing sampling techniques which process only around 1% or 2% of all dynamic memory instructions without unduly sacrificing the accuracy of interpolated results, execution time of instrumented code to gather memory address data from large parallel applications is prohibitive. One might naively hope that if one is only instrumenting and capturing 1% of all memory instruction slowdown would be minimal. There are are several reasons for this not being the case but most significantly note that ATOM and PIN do not allow one to remove snippets when sampling is off, one can only short-circuit the processing in
the snippets (return without doing anything). Dyninst, that could allow one to remove snippets, has by far the worse overheads to begin with. There are several key factors that cause an instrumented binary application to run longer than the original application.

1. Instrumented code executes more instructions than does the original binary (i.e. the instructions in analysis snippets in addition to the original instructions).

2. Jumping to an analysis snippet is a control flow interruption.

3. Program state has to be saved when jumping to analysis snippets.

4. Analysis snippets pollute cache from the standpoint of the original program.

5. Different amounts of processing in analysis snippets on parallel codes becomes a source of load-imbalance.

Factor 1 is an inherent artifact of gathering data using software-based techniques. The slowdown from factor 2 is related to the frequency and availability of data that one wants to gather and can be sometimes optimized as discussed in section 2.4. Factors 3 and 4 are amenable to optimization and this is investigated in section 2.3. Factor 5 has not been observed as a problem in our current efforts, but is a topic for future work.

Next we investigate the basic costs of binary instrumentation required to gather memory trace information. For the results in the following subsections Table 2.2 shows the experimental setup of each system. The slowdowns presented in this section are compared to the execution times of uninstrumented applications.

### 2.2.1 Measuring Overhead: Control Flow Interruption

Modern processors rely heavily on pipelines to achieve high instruction throughput. Once these pipelines bubble or break, performance deteriorates very quickly. When analysis snippets are inserted before every memory instruction in
an otherwise well-scheduled binary, the practical effect is to interrupt the pipeline every three or four instructions – a varying but usually large source of overhead depending on the architecture of the machine. In order to examine the cost of interruption of control flow, we exhibit Experiment 0 – which does nothing but insert a dummy analysis function before each memory reference. This dummy function is passed no parameters and merely returns. Thus we insert the fewest additional instructions possible for address acquisition into the original application, and induce no state-saving overhead (at least theoretically).

The Experiment 0 column of Table 2.3, Table 2.4, and Table 2.5 show the impact of interrupting control flow for every memory reference in a set of benchmarks. Slowdowns of around 10-fold are observed for both PIN and ATOM. The slowdown for Dyninst is significantly higher – due to a more onerous process for switching to analysis routines. DyninstAPI requires at least two jump instructions to install one snippets.

This experiment shows that the best one could hope to achieve in capturing all memory accesses dynamically is a 10-fold slowdown using these APIs, if every memory instruction is to be instrumented. Achieving the lower bound requires the analysis routine to be passed no parameters and for it to actually do nothing. In the next subsections we investigate the (worsening) effects of removing these restrictions.

2.2.2 Measuring Overhead: Maintaining Machine State

Any binary instrumentation tool must maintain correctness of the host application during an instrumented run. Thus, instrumentation tools normally wrap analysis snippets with safeguards when they are inserted into the host binary. These safeguards are responsible for saving and restoring the machine state that could be changed by inserted code snippets. Such overhead is frequent for fine-grained instrumentation (such as for each memory instruction) and can be quite expensive per instance, especially if the processor has many registers. Theoretically
it is sufficient to only save the registers that could be changed by the inserted analysis code, but APIs differ in their ability to optimize register saves.

These APIs, almost uniformly and except in trivial cases, simply save all the registers that could be touched in the snippets whether the snippets actually change them or not. Because of this, for the users of instrumentation tools, it is important to make the most visited snippets as simple as possible so fewer registers are required to be saved and restored. As an example, a call to a complicated cache simulation analysis snippet in the ATOM API requires saving all 64 registers before the instrumentation point and another 64 loads after. A similar effect is seen with Dyninst on Power 4 architectures, but an increased overhead is seen with PIN on Itanium architectures, due to larger register files.

The minimal impact that state saving can have on a memory address tracing tool is explored with Experiment 1. In this experiment, a one line analysis snippet is called before each memory access which records the effective address in a global array. A second routine is called at the end of each basic block to clear the global array. Effectively all addresses are recorded and then discarded shortly thereafter. This is a minimal overhead representative of recording memory addresses. The costs are shown in Tables 2.3, 2.4, and 2.5. It can be seen that using ATOM or PIN, the overhead caused by register saves and restores increases slowdown dramatically over Experiment 0 – ranging from 22 to 70 fold. Interestingly the overhead when using Dyninst did not increase much; Dyninst saves and restores all registers for all analysis snippets, no matter how simple.

2.2.3 Measuring Overhead: Cache Interference

Because the execution of analysis snippets and original application is interleaved, there could be contention between them in both the instruction and data cache. Normally this is minimized if the analysis snippets are very simple or if cache-friendly applications are being profiled. However, HPC applications of interest already stress the memory hierarchy, while the code to perform cache
simulations in analysis snippets accesses large data structures and has significantly complicated control flow. Consequently some of the instructions and data belonging to the host application may be evicted. These victims will have to be loaded into caches when the host application resumes execution. If this happens frequently enough, the performance of the instrumented code will drop dramatically.

Experiment 2 is presented to show the impact on runtime using full analysis code to collect effective addresses and run each through a set of 26 cache simulators. Only 2 runs were able to complete due to the incredible slowdown of this process. Note that sampling and a maximum visit to any given memory address was used in this experiment. That is, only when in a “sampling on” period would the memory access analysis actually run the cache simulators. Additionally, after a given memory access had been simulated 15,000 times, it was deemed no longer eligible for simulation.

Finally, it is important to note that the slowdown reported in Experiment 2 is not entirely due to cache interference. This experiment increases the number of instructions executed in the analysis snippets significantly (by running the cache simulators). However, it is this execution of extra instructions that is inherently tied to the increase in cache interference (it is hard to decouple the two effects).

<table>
<thead>
<tr>
<th>Machine</th>
<th>Processor</th>
<th>Tool version</th>
<th>Compiler and Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lemieux</td>
<td>Alpha EV67, 1GHz</td>
<td>ATOM V3.25</td>
<td>mpi77 -g3 -O4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cheetah</td>
<td>Power4, 1.33GHz</td>
<td>Dyninst 4.0</td>
<td>mpxlf -g -O5 -qstrict -qarch=ppwr4 -qtune=pwr4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Teragrid</td>
<td>Itanium2, 1.3GHz</td>
<td>PIN 1.71</td>
<td>mpir90-g -O3</td>
</tr>
</tbody>
</table>
Table 2.3: Instrumentation Overhead using ATOM:
execution time in seconds, slowdown in parenthesis

<table>
<thead>
<tr>
<th>App.</th>
<th>w/o Instru.</th>
<th>Experiment 0</th>
<th>Experiment 1</th>
<th>Experiment 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG.A.4</td>
<td>1.7</td>
<td>24.2 (14.2)</td>
<td>91.0 (53.5)</td>
<td>4857 (2857)</td>
</tr>
<tr>
<td>FT.A.4</td>
<td>9.0</td>
<td>89.0 (9.9)</td>
<td>316.1 (35.1)</td>
<td>12314 (1368)</td>
</tr>
<tr>
<td>MG.A.4</td>
<td>5.2</td>
<td>56.8 (10.9)</td>
<td>211.2 (40.6)</td>
<td>--</td>
</tr>
<tr>
<td>LU.A.4</td>
<td>47.0</td>
<td>731 (15.5)</td>
<td>2494 (53.0)</td>
<td>--</td>
</tr>
<tr>
<td>SP.A.4</td>
<td>84.0</td>
<td>769 (9.2)</td>
<td>2637 (31.4)</td>
<td>--</td>
</tr>
</tbody>
</table>

Table 2.4: Instrumentation Overhead using PIN:
execution time in seconds, slowdown in parenthesis

<table>
<thead>
<tr>
<th>Application</th>
<th>w/o Instrument</th>
<th>Experiment 0</th>
<th>Experiment 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG.A.4</td>
<td>1.3</td>
<td>22.2 (17.1)</td>
<td>112.6 (86.6)</td>
</tr>
<tr>
<td>FT.A.4</td>
<td>5.4</td>
<td>61.2 (11.3)</td>
<td>279.7 (51.8)</td>
</tr>
<tr>
<td>MG.A.4</td>
<td>2.4</td>
<td>28.0 (11.7)</td>
<td>138.0 (57.5)</td>
</tr>
<tr>
<td>LU.A.4</td>
<td>27.8</td>
<td>406.8 (14.6)</td>
<td>1848.0 (66.5)</td>
</tr>
<tr>
<td>SP.A.4</td>
<td>55.9</td>
<td>602.8 (10.8)</td>
<td>2664.0 (47.7)</td>
</tr>
</tbody>
</table>

Table 2.5: Instrumentation Overhead using DyninstAPI:
execution time in seconds, slowdown in parenthesis

<table>
<thead>
<tr>
<th>Application</th>
<th>w/o Instrument</th>
<th>Experiment 0</th>
<th>Experiment 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG.A.4</td>
<td>2.2</td>
<td>1920.0 (872.7)</td>
<td>1933.9 (878.6)</td>
</tr>
<tr>
<td>FT.A.4</td>
<td>7.2</td>
<td>7302.3 (1000.3)</td>
<td>7327.0 (1003.7)</td>
</tr>
<tr>
<td>MG.A.4</td>
<td>5.1</td>
<td>4716.9 (924.7)</td>
<td>4756.5 (932.5)</td>
</tr>
<tr>
<td>LU.A.4</td>
<td>65.4</td>
<td>&gt;19720 (301.4)</td>
<td>--</td>
</tr>
<tr>
<td>SP.A.4</td>
<td>96.8</td>
<td>&gt;19720 (203.7)</td>
<td>--</td>
</tr>
</tbody>
</table>

2.3 Reducing Overhead Caused by Saving Machine States

The big differences between Experiment 0 and Experiment 1 using ATOM and PIN are mostly caused by saving and restoring system states. Such overhead can be improved with a simple technique – buffering of memory addresses before processing them\(^1\). This effectively reduces the impact of state saving overhead by having the simplest possible analysis snippet for each memory access.

\(^1\)This is also referred to as asynchronous processing, in contrast to the term asynchronous recording introduced in Chapter 3
A memory access probe was implemented in a single line of C code, to store the effective address of a memory access in a large global buffer. The more complex cache analysis was delayed until such time as the buffer is full. In the less frequently invoked basic block level analysis snippet (which controls sampling among other things) the effective address buffer is monitored and run cache simulators will be invoked when the buffer is full.

With this much simplified per memory access snippet, the overhead of state saving for the most frequent analysis snippet has been greatly reduced. Since the cache simulator code is run through the less frequent basic block analysis snippet, one sees a decrease in state saving overhead proportional to the number of static memory accesses per basic block.

Additionally, this approach ameliorates cache interference. The primary offender in this arena is the cache simulation code, which simulates 26 cache structures for each memory access. By buffering a large number of effective addresses and then running the cache simulators for all of those addresses at once the frequency of interference with the program’s cached data is reduced. Here a buffer of 10,000 addresses is implemented, reducing the number of times the original application’s cache behavior is impacted significantly. Overall, using ATOM, one finds buffering to speedup unbuffered tracing by as much as 30 fold.

### 2.4 Reducing Instrumentation Points

The results of Experiment 0 indicates that if all the memory instructions are to be instrumented, even with a dummy function, the minimum overhead is more than 10 fold. This is an impractical lower bound for any large applications one is interested in. Most of the available supercomputers have a queue limit of less than 12 hours. Originally checkpoints and multiple traces were applied to overcome this limit. During every run, only a small subset of basic blocks were instrumented and traced. That was a painful process. It normally takes weeks, if
not months to finish tracing one whole application this way. In addition, partial tracing can introduce new noise and errors into the simulation.

In order to reduce the overhead to an acceptable level, all memory instructions cannot be instrumented at the same time! Fortunately one can collect full memory address by only instrumenting a subset of memory instructions. Many memory instructions are in fact related. Such relationships can be detected through static analysis and used to reduce the number of instrumentation points required to get complete effective addresses.

2.4.1 Chaining Memory Instructions

With simple static analysis of the binary code in each basic block, we chain memory instructions together if they will always keep the same static offset during run-time. Only one of the memory instructions in a chain (the leader) must be instrumented. The others’ effective addresses can be calculated from the leader’s address.

Consider a basic block code segment shown in Figure 2.1 where there are 7 memory instructions: 0,1,4,5,8,9, and 10. Note that lda on Alpha is not a memory instruction (rather it is like an addi). Until execution, we do not know the value of the registers so there is no way to tell the exact effective address of these memory instructions. But we can tell from static analysis that the effective address of instruction 1 will always be eight less than the effective address of instruction 4. The effective address of instruction 9 will always be 8 less than that of instruction 10. Using this knowledge, we do not need to instrument instruction 4 and instruction 10 if instructions 0 and 9 have already been instrumented. However, we are not as successful in chaining instruction 0 to 9 and 10 due to the modification of the shared register $9 in instruction 3.

The new algorithm first groups memory instruction if the base register used in these memory instructions are guaranteed to be the same. Those memory instructions using two registers for addressing register-register base register plus
displacement addressing, are put into groups of their own. After we have put all the memory instructions in the block into groups, a candidate from each group will be selected to be instrumented with StoreAddr(). This candidate is assigned a unique id. When the candidate is instrumented, the id is passed as one argument in addition to the effective address. During runtime, StoreAddr() will put effective addresses and group ids into a buffer. Figure 2.2 gives the algorithm. Pre-execution static analysis groups memory accesses into chains and elects leaders that will be instrumented. Data on chain membership is imported into the analysis routine and used to direct cache simulation when a leader is dynamically referenced.

Chaining memory instructions can reduce the number of instrumentation points, but still leaves a number of control flow interruptions (at memory accesses) that impair performance. Additionally, register values which are updated within the basic block break chains and create additional instrumentation points.
for all the instruction in block
{
    if it is a memory instruction
       using $r$ as base register
       {
           1. put this memory into the
               chain of $r$ record offset
               from the instruction
           2. record the position of
               this instruction in the block
       }
    if it is a memory instruction using
        "register register" addressing
        1. create a chain of its own

    if it writes into a regular register $r$
        if there are memory instructions
           using $r$ before
           {
               1. export the memory
                   instructions in the chain
               2. empty the chain
           }
}
for each non-empty chain, export the memory
instructions in these chains.

Figure 2.2: Algorithm for Chaining Memory Instructions

Exporting a memory instruction will write chain id, new offset, original
instruction index, etc. into a file

2.4.2 Delayed Instrumentation with Symbolic Execution

Chaining memory instructions can reduce the number of instrumentation
points, but there is still one instrumentation point per chain. These instrumen-
tation points leave a number of control flow interruptions (at memory accesses) that
impair performance. Additionally, register values which are updated within the
basic block break chains and create additional instrumentation points. The goal
of delayed instrumentation is to find a way to merge all memory address collection into a single instrumentation call at the end of the basic block.

The key is to record register values rather than the effective addresses themselves. These register values can then be used in conjunction with static knowledge of the basic block binary, such as immediate offsets, to enable one to re-create the effective addresses of each memory instruction in the basic block. Straightforward delayed instrumentation would, in the case of the code in Figure 2.1, allow one to remove instrumentation points before instructions 1, 8 and 9 and instead use a single instrumentation call at the end of the basic block to save the values of register $13$, $19$ and $9$. The effective addresses of these memory instructions can be properly calculated from the recorded corresponding register values. Because a per basic block analysis routine already exists, we simply increase the number of parameters passed.

Removing instrumentation before instruction 0 requires a bit more effort. The recorded value of register $9$ is not the exact value when instruction 0 executes. Register $9$ is modified by a lda instruction in the middle of the basic block (which originally broke $9$’s references into two chains). We notice that lda is making a simple modification to register $9$ and the effect can be reversed to deduce the correct value for instruction 0.

We define a set of tractable modifications to registers which we can easily reverse the effect of, and calculate earlier register values based on, a final end-of-basic block register file snapshot. Table 2.6 lists the four integer instructions we support as well as the symbolic execution that updates register values. Any other instructions that assign to an integer register are defined as intractable. If and only if a register used for addressing is modified by an intractable instruction in the middle of the basic block, we insert a snippet to store its value before modification. We cannot remove the instrumentation point before instruction 5 because register $19$ has been changed intractably. For the block in Figure 2.1, we only need to explicitly instrument one memory instruction, the rest are captured
at the end of the basic block.

Table 2.6: Instructions Supported by Symbolic Execution

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Symbolic Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>lda</td>
<td>lda ( \text{target, imm(base)} )</td>
<td>( \text{target} = \text{base} + \text{imm} )</td>
</tr>
<tr>
<td>ldah</td>
<td>ldah ( \text{target, imm(base)} )</td>
<td>( \text{target} = \text{base} + \text{imm} \times 65536 )</td>
</tr>
<tr>
<td>addli, addqi</td>
<td>addi ( \text{target, base, imm} )</td>
<td>( \text{target} = \text{base} + \text{imm} )</td>
</tr>
<tr>
<td>subli, subqi</td>
<td>subq ( \text{target, base, imm} )</td>
<td>( \text{target} = \text{base} - \text{imm} )</td>
</tr>
</tbody>
</table>

Figure 2.3 shows an excerpt from a very common assembly sequence on an Alpha architecture from an important basic block in the NAS Parallel Benchmark SP. Note that it seems there are four registers which are used to address into memory for load instructions. In reality these registers are serving in the role of temporary variables used to construct a base plus displacement style address where the displacement is larger than 16 bits in size. These are actually 4 accesses to different displacements from the address in register $\text{\$7}$. These temporary registers are frequently assigned in the block. Such assignments make delayed instrumentation less effective. However, based on the fact that the values of these temporary registers are calculated from register $\text{\$7}$, the effective addresses of memory instructions using these temporary registers can also be calculated using register $\text{\$7}$. For example, register $\text{\$8}$ is changed in instruction 17, which makes the register value used for addressing by instruction 13 and 17 intractable. One the other hand, the value of register $\text{\$8}$ at instruction 13 equals to the value of register $\text{\$7}$ at instruction 4 plus $65536 \times 4$. If $\text{\$7}$ is tractable and can be delayed, the effective addresses of instruction 13 and 17 can be calculated from the end-of-block value of register $\text{\$7}$, plus proper adjustment.

Identifying and optimizing this form of register aliasing provides another chance to delay instrumentation. It has been shown to greatly increase the ability to remove instrumentation points.

In implementation, we try to delay all the register recordings to the end of the basic block, so the algorithm starts from the last instruction in the basic block
and goes backwards. We have an array of adjustment values for each register and all the adjustment are initialized to zero. Every time an instruction writes into a register, we will update the adjustment value of the target instruction by symbolically execution. If the instruction is not one of the instructions we can symbolically execute, the adjustment value will be specially marked. For example $4$ is assigned in instruction 8 in the log, it is not possible to tell what its value is just before instruction 7, so a StoreAddr() is inserted right before instruction 7. Every time we meet a memory instruction, if the adjustment value of the register is not the specially marked value, the offset of the memory instruction is updated based on the adjustment value. Otherwise it will be instrumented immediately to record the effective address.

The algorithm scans the instructions in the block twice. During the first, it scans the instructions in the block from top to bottom to identify memory instructions and to identify register aliasing. The second time the instructions are scanned in reverse order to determine whether the base register value of each memory instruction can be deduced from register values recorded at the end of the block. If the register value has been changed intractably, a StoreAddr() will be inserted right before the memory instruction. Once a StoreAddr() is inserted in the middle of the block, we know any remaining memory instructions using the same register can no longer use what is recorded at the end of the basic block. However

```
I  1 ldah $25,18($7)
I  3 ldah $27,9($7)
I  4 ldah $8,4($7)
I  8 ldah $24,18($7)
M  13 ldt $f16,25352($8)
M  16 ldt $f15,-29536($24)
M  17 ldl $8, 9($8)
M  21 ldt $f11,25480($27)
M  22 ldt $f14,10648($25)
```
this StoreAddr() can be used as a leader. The rest of the memory instructions using the same register are put into to the same group as this memory instruction. Their effective addresses are calculated from what is recorded in StoreAddr()

Figure 2.4 lists the simplified algorithm without dealing with register aliasing. Delaying and merging instrumentation through basic symbolic execution

from the last instruction to the first in one basic block
{
  if Inst in {lda, ldah, addi, subqi} and $source == $target
    UpdateRegValBySymbolicExe($target, $source, imm);
  else
    RegTable[$target].delayable = false;
    RegTable[$target].value = 0;
  if Inst is mem (such as ld $target, offset($source))
    if RegTable[$source].delayable == false
      Mark to insert StoreAddr() to store Eff Address;
      RegTable[$source].delayable = true;
      RegTable[$source].value = (0 - offset);
    else
      Mark instruction is delayed;
      UpdateInstOffset(offset, RegTable[$source].value);

  Insert StoreAddr() at marked locations;

  Instrument at the end of the block to record all delayed register values;

from the first memory instruction to the last in the block
{
  Assign an id to one of the stored values
    (either StoreAddr() or delayed register values)
    it uses for address calculation
  Write its group id and updated offset to a file;
}

Figure 2.4: Simplified Algorithm of Delayed Instrumentation

Register Alasing is not processed in this algorithm

reduces the number of instrumentation points significantly. We may have very few
instrumentation points in the main body of a basic block, thus there is much less interruption of control flow of the original application.

An additional performance benefit is that less data must be buffered when storing register values than when storing effective addresses. This results in reduced numbers of interruptions from the analysis routine which processes the buffer.

2.5 Regenerate Addresses from Recorded Register Values

For each memory instruction, we need to record its new offset and the id of the stored register value so the actual effective address can be calculated from them at runtime. The id of the register value is the sequence number of recorded dynamic values for that block. For example, in the instrumented block in Figure 2.5, there is a StoreAddr() inserted before instruction 7 and it comes first before any other instrumentation points (BlockReference3 at the end of the block), so the point index of that instrumentation is zero. After the blocks have been instrumented, information used for regenerating the effective addresses are exported into a file. This file is read in when the instrumented binary starts to execute. When the address buffer is processed, addresses are generated in the same order as they appear statically in the basic block. The values are calculated using the new offsets and corresponding register values recorded in the buffer. Although the register values recorded in the buffer are not in the same order as how they are used by the memory instructions, the effective addresses are generated in the exact order as they will appear during execution. For example, the register value used by instruction

\[ \text{ldt } \text{f10}, 0(\text{f4}) \]

is the first value of the block recorded in the buffer, the effective address of this memory instruction won’t be generated until the previous two have been generated and processed. The effective address of the second memory instruction
Figure 2.5: Instrumentation Log

Example shows how the basic block is instrumented with delayed instrumentation technique. Numbered lines are original instructions. Point index starts from zero, which is the same order as the values are recorded into the buffer.

1, ldt $f1,0($2)

is $-8$ plus the third value (for register $2$) from current position in the address buffer as indicated in Figure 2.5.
2.6 Related Issues

In order to make sure that addresses can be generated correctly during runtime, we require there is no control transfer in the basic block. A "Basic Block" has to be a single entry and single exit section in the binary. There should not be any function calls in the block either, otherwise the memory instructions before and after can be mistakenly grouped. If the instrumentation tool does not guarantee this requirement, we need to break the code section before every possible control transfer and insert BlockReference() before such control transfer.

Exceptions that happen in the middle of a block could also cause problems, especially if the exception handling functions are also instrumented. The analysis routine will lose synchronization in generating the effective addresses from recorded register values. However most HPC applications rarely rely on exceptions in the program. Exceptions generally happen if there are bugs in the code. If there is a bug in the application causing exceptions, the reports from on-line processing could be wrong and it is the programmer’s responsibility to first correct these bugs.

Currently the implementation is not thread safe. However once the address buffer and related variables are made local to each thread, the scheme works on multi-thread platforms. We did not implement this scheme because all the applications we are interested are MPI programs, which only have one thread(process) per processor.

Buffering has been shown to be very effective in improving trace performance. Generally the bigger the buffer, the better the performance. However for parallel applications, instrumentation can cause extra overhead due to synchronization. When the buffer is too big, each processor takes much longer in the analysis routines – sometimes it causes busy waiting of other processors. Currently the buffer is set to 10,000 entries and such busy waiting is negligible.

We anticipate that each of these techniques must re-analyzed when considering porting them to other binary instrumentation frameworks. In particular,
the chaining approach provides no improvement for handling register-register addressing, which is commonly used for indirect memory references on Power processors because the distances of effective addresses of such instructions cannot be determined prior to execution.

The efficiency of using simple probes to reduce the costs of saving machine states depends on the ability of the instrumentation tools. We see in Table 2.5 that there is little difference between Experiment 0 and Experimen 1. So using simpler probes is practically useless when one uses DyninstAPI. Apparently DyninstAPI lacks the ability to find the registers which are not used by the inserted code snippets so it saves all 64 registers no matter what at each instrumentation point.

2.7 Results

The static analysis algorithms were implemented, integrating chaining and delayed instrumentation into ATOM based Metasim, and then the speed on Lemieux, an Alpha based system at the Pittsburgh Supercomputing Center (PSC), was measured.

Table 2.7 shows the number of actual memory address instrumentation points required after using the static analysis and delayed instrumentation techniques. One sees that less than 10% of total memory instructions in most applications are required to be instrumented. Although fewer instrumentation points generally leads to faster tracing, the actual speed depends on whether the most frequently executed sections of code have significantly fewer instrumentation points. For example, E3D only shows a reduction of 80% in number of instrumentation points after static analysis, but the 12 basic blocks which account for more than 90% of total memory references have had all memory instrumentation points optimized away. All 925 memory instructions in those blocks can be analyzed with delayed instrumentation at the end of their basic blocks. From Table 2.10, one can see the slowdown of E3D has been reduced to only 4-fold despite the many
remaining instrumentation points.

Table 2.7: Reduction in Number of Instrumentation Points with Delayed Instrumentation

<table>
<thead>
<tr>
<th>Application</th>
<th>Num. of Mem Inst</th>
<th>Delayed Instrumentation Points</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG.A.4</td>
<td>3595</td>
<td>141</td>
<td>96.1%</td>
</tr>
<tr>
<td>FT.A.4</td>
<td>4302</td>
<td>224</td>
<td>94.8%</td>
</tr>
<tr>
<td>MG.A.4</td>
<td>9662</td>
<td>439</td>
<td>95.5%</td>
</tr>
<tr>
<td>LU.A.4</td>
<td>20579</td>
<td>233</td>
<td>98.9%</td>
</tr>
<tr>
<td>SP.A.4</td>
<td>38383</td>
<td>306</td>
<td>99.2%</td>
</tr>
<tr>
<td>HYCOM</td>
<td>87520</td>
<td>3029</td>
<td>96.5%</td>
</tr>
<tr>
<td>AVUS</td>
<td>124341</td>
<td>10013</td>
<td>92.0%</td>
</tr>
<tr>
<td>E3D</td>
<td>45698</td>
<td>9373</td>
<td>79.5%</td>
</tr>
</tbody>
</table>

Next assessed is how well the technique of static analysis and delayed instrumentation impacts the identified costs of control flow interruption (Experiment 0) and costs of saving machine states (Experiment 1). Again, Experiment 0 uses dummy instrumentation functions that do not actually record any data and Experiment 1 actually collects memory addresses, but immediately discards them without running any further processing.

Table 2.8 shows that the overhead of simply instrumenting to gather necessary effective addresses can be reduced from about a 10 fold slowdown to a range of 1.2 to a 3.5 fold slowdown. Adding in a bit more realism, Experiment 1 in Table 2.9 (which actually collects effective addresses) shows that for simplest processing of the collected addresses (by acquiring and discarding them), one could achieve slowdowns of less than 6 fold and often around 2-3 fold.

Table 2.10 compares the execution time of the memory trace code (Metasim) running 25 cache simulators using ATOM, showing the impact of the two steps of the static analysis optimizations. The buffering column reports the baseline execution time using sampling and a 10,000 element buffer. The chaining column baseline shows that speedup from chaining is very application dependent with CG.A.4 showing only a 1.07 times speedup over buffering while SP.A.4 shows over
Table 2.8: Comparison of Instrumentation Overhead: Experiment 0
slowdown in parenthesis

<table>
<thead>
<tr>
<th>Application</th>
<th>w/o Instru.</th>
<th>Original</th>
<th>Delayed</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG.A.4</td>
<td>1.7</td>
<td>24.2 (14.2)</td>
<td>5.8 (3.41)</td>
</tr>
<tr>
<td>FT.A.4</td>
<td>9</td>
<td>89.0 (9.9 )</td>
<td>13.8 (1.53)</td>
</tr>
<tr>
<td>MG.A.4</td>
<td>5.2</td>
<td>56.8 (10.9)</td>
<td>7.5 (1.44)</td>
</tr>
<tr>
<td>LU.A.4</td>
<td>47</td>
<td>730.7 (15.5)</td>
<td>56.5 (1.20)</td>
</tr>
<tr>
<td>SP.A.4</td>
<td>84</td>
<td>769.4 (9.2 )</td>
<td>120.7 (1.44)</td>
</tr>
</tbody>
</table>

Table 2.9: Comparison of Instrumentation Overhead: Experiment 1
slowdown in parenthesis

<table>
<thead>
<tr>
<th>Application</th>
<th>w/o Instru.</th>
<th>Original</th>
<th>Delayed</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG.A.4</td>
<td>1.7</td>
<td>91.0 (53.5)</td>
<td>10.8 (6.35)</td>
</tr>
<tr>
<td>FT.A.4</td>
<td>9</td>
<td>316.1 (35.1)</td>
<td>19.0 (2.00)</td>
</tr>
<tr>
<td>MG.A.4</td>
<td>5.2</td>
<td>211.2 (40.6)</td>
<td>14.0 (2.50)</td>
</tr>
<tr>
<td>LU.A.4</td>
<td>47</td>
<td>2494.0 (53.0)</td>
<td>74.4 (1.58)</td>
</tr>
<tr>
<td>SP.A.4</td>
<td>84</td>
<td>2637.1 (31.4)</td>
<td>209.1 (2.49)</td>
</tr>
</tbody>
</table>

A 15 times speedup. The effectiveness of chaining is dependent on how many memory references can be chained in the important basic blocks. The results in the delay column show the significant impact of additional static analysis to enable delaying as many memory instrumentation points as possible. Slowdowns over the original uninstrumented code are shown in parenthesis and one can see they are much less than simply buffering. The larger slowdowns appearing in shorter applications are due to the fact they are too short for the sampling to be effective. Of specific note is that results on full applications such as HYCOM, AVUS, and E3D are particularly good. An additional benefit of this improvement is that these traces can now be collected in a single run and still fit within standard queue limits (typically 12 hours). Previously, the much higher slowdown of tracing required a multi-stage process where many applications were traced in 10 different runs – collecting data on subsets of basic blocks each run.
Table 2.10: Tracing Time Comparison in seconds (slowdown in parenthesis)

<table>
<thead>
<tr>
<th>App</th>
<th>w/o Instru.</th>
<th>w/ Buffering</th>
<th>Chaining</th>
<th>Delayed</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG.A.4</td>
<td>1.7</td>
<td>168 (98.8)</td>
<td>156 (91.8)</td>
<td>65 (38.2)</td>
</tr>
<tr>
<td>FT.A.4</td>
<td>9.0</td>
<td>398 (44.2)</td>
<td>298 (33.1)</td>
<td>166 (18.4)</td>
</tr>
<tr>
<td>MG.A.4</td>
<td>5.2</td>
<td>560 (107.7)</td>
<td>376 (72.3)</td>
<td>156 (30.0)</td>
</tr>
<tr>
<td>LU.A.4</td>
<td>47.1</td>
<td>3799 (80.7)</td>
<td>1605 (34.1)</td>
<td>813 (17.3)</td>
</tr>
<tr>
<td>SP.A.4</td>
<td>84.0</td>
<td>5676 (67.6)</td>
<td>376 (45.3)</td>
<td>1568 (18.7)</td>
</tr>
<tr>
<td>CG.B.8</td>
<td>67.2</td>
<td>1812 (27.0)</td>
<td>840 (12.5)</td>
<td>396 (5.9)</td>
</tr>
<tr>
<td>FT.B.8</td>
<td>70.3</td>
<td>1932 (27.5)</td>
<td>509 (7.2)</td>
<td>253 (3.6)</td>
</tr>
<tr>
<td>MG.B.8</td>
<td>8.9</td>
<td>864 (97.2)</td>
<td>543 (61.1)</td>
<td>221 (24.9)</td>
</tr>
<tr>
<td>CG.C.8</td>
<td>183</td>
<td>3967 (21.7)</td>
<td>1426 (7.8)</td>
<td>743 (4.1)</td>
</tr>
<tr>
<td>FT.C.8</td>
<td>189.6</td>
<td>17630 (50.4)</td>
<td>11652 (33.3)</td>
<td>7978 (22.8)</td>
</tr>
<tr>
<td>MG.C.8</td>
<td>63.5</td>
<td>3544 (55.8)</td>
<td>1129 (19.4)</td>
<td>480 (7.6)</td>
</tr>
<tr>
<td>HYCOM</td>
<td>3506</td>
<td>78855 (22.5)</td>
<td></td>
<td>19633 (5.6)</td>
</tr>
<tr>
<td>AVUS</td>
<td>2359</td>
<td>34816 (14.5)</td>
<td></td>
<td>11851 (5.0)</td>
</tr>
<tr>
<td>E3D</td>
<td>1569</td>
<td>34058 (21.7)</td>
<td></td>
<td>6388 (4.0)</td>
</tr>
</tbody>
</table>

2.8 Limitations

Memory tracing requires instruction level instrumentation. However currently available instrumentation tools are not efficiently designed to deal with these fine-grained instrumentation tasks such as instrumenting every memory instructions. Although tools like ATOM, PIN and DyninstAPI can be used for memory tracing, straightforward implementation using these tools causes severe slowdown. The root of these problems is that current tools treat each instrumentation point individually and introduce too many instructions around each instrumentation point.

One can use static analysis to reduce the number of necessary instrumentation points, and use buffering techniques to reduce costs of each instrumentation point. It is nevertheless very challenging to implement these techniques on top of generic binary instrumentation tools. For example Intel IA64 ISA is very complicated. It is almost impossible for a non-expert of this ISA to go through instructions and efficiently implement grouping and delayed instrumentation techniques.

In addition, static analysis isn’t always effective in reducing the number
of instrumentation points. For example CG’s most frequently visited two basic blocks, which account for over 90% of total dynamic memory references, have several indirect memory references. These indirect memory references prevent static analysis from grouping memory instructions or delaying instrumentation points thus resulting much higher overhead even after applying all the techniques. Static analysis can also be less effective on IA64 because of all the hidden rules such as register rotation, software pipelining etc. It is very complicated to assert whether the register values will be same or not when they can be changed implicitly. When it is uncertain about changes of register values, static analysis will leave many instrumentation points in the middle of the basic block, thus the instrumented binary will be dramatically slower.

Due to these difficulties and inefficiencies, a binary instrumentation tool with low overhead and less requirements on the user’s ability to parse the binary will be very desirable. Such a tool will be presented in the next chapter.

This chapter, in part, is a reprint of the material as it appears in “Reducing Overheads for Acquiring Dynamic Memory Traces”, by Xiaofeng Gao, Michael Laurenzano, Beth Simon and Allan Snavely, published in the proceedings of 2005 IEEE International Symposium on Workload Characterization (IISWC05) Oct.05 Austin [28]. The dissertation author was the primary researcher and author; the co-authors listed on these publications directed, supervised, or helped the research which forms the basis for these chapters.
Chapter 3

A Lightweight Asynchronous Tracing Scheme

This chapter introduces ALITER: an asynchronous lightweight instrumentation tool for event recording. This instrumentation tool serves two purposes in the PMaC performance modeling framework. First it is created to work on IBM POWER processors because we currently do not have a usable instrumentation tool for IBM processors. Secondly, and more importantly, it is implemented to demonstrate a lightweight asynchronous tracing scheme.

This tracing scheme specializes in handling tracing tasks that can be programmed asynchronously. It does not invoke the user’s routines every time an interesting event happens, as most current synchronous tools will do. Instead, the events are first stored in a buffer created in the instrumentation tool. The user provided analysis routines are invoked only when the buffer is full. Such an asynchronous event recording scheme can dramatically reduce the time overhead of memory tracing. Overall less than 2 fold slowdown has been measured on the selected benchmarks to collect memory traces. For common cases, only one instruction is needed for each instrumentation point, plus 20 extra instructions per each basic block, which are responsible for maintaining the status of the buffer. Comparatively one will have tens of fold slowdown using ATOM and PIN and
nearly 1000 fold slowdown using Dyninst for accomplishing the same tasks.

3.1 Limitations of Synchronous Tools

Most binary instrumentation tools use a synchronous implementation – each interesting event is captured and immediately passed to a user-provided function for processing. In such a synchronous instrumentation scheme, each captured event requires the user’s immediate intervention and response, whether such response is to pass the event through a full simulator, to put it in a buffer, or to simply discard it. In binaries instrumented with these tools, capturing events and processing them are essentially inseparable.

Synchronous instrumentation tools guarantee the user’s ability to process each event immediately after the event occurs. Such ability is extremely valuable for handling time-critical tasks such as debugging or detecting security intrusions. However, it is considerably more expensive to have a function call after each interesting event happens, especially if these events, such as memory accesses, happen frequently. It has been shown in previous chapters that inserting a dummy function before each memory instruction makes instrumented binaries run at least 10 times slower. Inserting more complicated analysis routines will cause even more slowdown.

On the other hand, many instrumentation tasks are not time-critical; therefore there is no need to process each event immediately after the event happens. For example, most trace-driven simulators could collect event traces and process them asynchronously and still be correct. In other words, collecting the events is logically independent from the actual simulation of traced events. In fact, for asynchronous tasks where prompt response is not necessary, immediately processing each individual event through more complicated simulators should be discouraged due to adverse effects on cache performance and other extra costs that have been shown in previous chapters.
Buffering techniques were introduced in previous chapters to effectively turn synchronous processing into asynchronous processing using available synchronous tools: addresses are captured and stored in user-provided buffers via simpler probes before they are passed to simulators for more complicated processing such as actual simulation.

Although asynchronous processing techniques using synchronous tools have been quite successful in reducing time overhead for event tracing, event recording is still synchronous and causes significant overhead. Here an example is presented to highlight the limitations of buffering techniques with synchronous tools: assume that there are interesting events in the basic block shown in Figure 3.1 that one wants to capture before instructions 3, 5 and 7. A simple probe called RecordEvent() is provided for capturing an event and storing it into a buffer. The probe will call ProcessBuffer(), a function that processes all events previously stored in the buffer, once the buffer is full.

With synchronous instrumentation tools, the users direct the tool to insert function calls to RecordEvent() before instructions 3, 5 and 7. The instrumentation tool will then internally wrap these instrumentation points with several store/load instructions before actually issuing call instructions and instructions for parameter passing, so that inserted calls will be transparent to the original instructions. Figure 3.2 shows the final instrumented instructions using synchronous
tools like ATOM.

Closer examination will reveal that many instructions will be executed for recording each single event. First are the call instruction and surrounding pairs of load/store instructions for saving and restoring special registers and volatile registers. On an IBM Power4 processor, at least 12 registers absolutely must be saved before each inserted call instruction and restored afterward due to the call conventions on this platform. On other machines the number of registers required to be saved is similar. These extra load/store instructions cannot be removed as long as there are call instructions inserted into the binary.

Next, RecordEvent will have to execute some instructions to write the captured event into the event buffer. Typically, the following actions will have to be taken every time the function is invoked:

1. Load the event buffer size from a memory location.
2. Compare the buffer size to some threshold.
3. Calculate the memory address of the next entry in the buffer.
4. Write the event to the calculated memory location.
5. Increment buffer size by one.

6. Update the memory location where buffer size is stored.

That is a lot of work to simply write a value into a buffer! It is even more so if there are many such calls inserted into the binary, which is exactly the case for memory tracing.

It should be clear that implementing asynchronous tracing cannot be done efficiently with current synchronous instrumentation tools. Besides the obvious costs caused by call instructions and accompanied load/store instructions, synchronous tools make it impossible for users to take advantage of the fact that only action 4 in the list above is unique and absolutely necessary for each event recording. These tools try all means to encapsulate each instrumentation point so user provided functions can correctly do whatever they want with the captured events. Unfortunately, encapsulation prevents common actions from being merged and reduced; they will have to be repeated at each instrumentation point. For example, buffer size will be compared to the threshold at every event capture point, even if the buffer is far from being full.

### 3.2 An Asynchronous Instrumentation Scheme

Based on the analysis above and the fact that asynchronous profiling and tracing is of great interest, a special lightweight asynchronous instrumentation scheme is proposed in this chapter to dramatically reduce the costs of event recording. The two causes of inefficiency identified above are targeted in this new scheme. First a buffer is created by the instrumentation tool to keep recorded events temporarily before passing them to user’s analysis routines for processing. No call instructions are used for recording events into the buffer; instructions for such purposes are all inlined. Besides the elimination of control hazards caused by call instructions, there is practically no need to save and restore any special and volatile registers. Furthermore, inlined instructions now can be scheduled together
with original instructions so the resulting binaries can better utilize the processor’s instruction level parallelism mechanisms.

Secondly, event recordings are managed at the basic block level instead of treating them as individual tasks. Common actions are merged, thus there are far fewer instructions required for event recordings. Since the number of events in each basic block is statically known by instrumentation tools, the buffer size needs to be checked and updated only once per basic block. The calculation to determine where in memory the next event should be saved also needs to be done only once every basic block; one single store instruction with a pre-determined offset is all it takes to write an event into the buffer. These savings could become dramatic when there are many events to record in the blocks because otherwise such work has to be done for recording of each event in that block.

Figure 3.3 shows what the instrumented instruction mix will look like if an asynchronous instrumentation tool is used to accomplish the same tracing tasks as shown in Figure 3.2. A prologue and an epilogue will be created for each basic block to cover all common tasks such as saving registers, checking buffer
size, etc. Recording an event is accomplished with a single store instruction that has a pre-determined offset. There are indeed many instructions in the prologue; however the number of instructions in each prologue is about the same as the number of instructions that synchronous tools will normally have at each event. The overall savings in dynamic instruction counts is proportional to the number of events one wants to record in each basic block. Particularly for tracing memory accesses, which normally happen many times in a basic block, such asynchronous instrumentation schemes can reduce tracing costs dramatically.

Synchronous instrumentation tools require tool users to go through the instructions and determine where and how to insert functions so events can be captured and recorded correctly. Such tasks are tedious and error-prone. For example, it is the user’s responsibility to decide whether a function call should be inserted before or after an instruction. Such decisions could make big differences on what are eventually captured. With an asynchronous instrumentation scheme, many of the responsibilities have been shifted from the tool users to the tools. It is now the instrumentation tools’ responsibility to check the buffer size and write an event into the proper location in the buffer. It is also the tools’ responsibility to call user provided simulators once the buffer is full. All the tool users have to do is to provide a function which will process all the elements in the event buffer.

ALITER is such an instrumentation tool implemented based on the asynchronous scheme described above on the IBM POWER4 platform. In next section, the major implementations issues and many details of ALITER will be explored and explained.

### 3.3 Implementation and Optimizations

ALITER is implemented as an assembly rewriter instead of as a binary rewriter because it is easier to analyze and modify assembly code. Some tedious tasks, such as offset calculation can be done by the assembler. However, the tech-
The techniques presented in this section can be implemented in the style of a binary rewriter that writes a new instrumented binary, such as ATOM. This section illustrates how one can collect memory traces with asynchronous instrumentation. For other types of traces such as call graph determination, a similar method can be used. However, the possible benefits of asynchronous instrumentation are more dramatic the finer-grained the trace task.

One first compiles the source files to assembly codes, with the same optimization flags one would use to compile the executable directly. The assembly codes are then parsed to break the instructions into basic blocks. The instructions in each basic block are analyzed to determine the necessary instrumentation points. In this example, all the memory instructions are identified to determine how to instrument them. ALITER uses the chaining method discussed in [28]. Memory instructions are divided into chains based on their register values. Only one instruction in each chain is selected to be instrumented. In addition, to keep the extra instruction(s) simple, only the register values are stored. The static offset in the instructions are exported to a file and used to re-generate the effective addresses. The process by which the tool instruments memory instructions is transparent to the user. The user can acquire all the memory instructions one by one using the tool provided interfaces. If, in the future, the tool utilizes a different buffer organization, the user does not have to modify the analysis routine.

Based on these experiments, almost all basic blocks have at least one GPR (general purpose register) that is not used (read or write) in the block. One can identify that register and use it to point to the location in the buffer where the values should be stored. Such a register is referred as “regbase” in the following discussion. To store the register values used by one memory instruction, the instrumentation tool simply issues one instruction which writes the register value to a location in the address buffer using regbase right before the original memory instruction. For instructions using two registers to address memory, two such store instructions could be needed, depending on the outcome of the chaining
technique. Regbase is not used by the original instructions so it maintains the pointer to the buffer throughout the block.

For each instrumented block, ALITER inserts a prologue before the first instruction and an epilogue before the last instruction. The prologue saves the original value of regbase. It also updates the value that indicates how many items have been stored in the buffer and calculates the new start position in the buffer for storing the values for this block. The size of the prologue is about 10 instructions on Power4 systems. To make the instructions for buffer maintenance efficient, more registers are needed. In reality for many basic blocks, there are registers assigned in the block before they are used. The values of such registers are dead when execution enters the block. One is free to use these registers before they are assigned without having to save their values. It is easier to detect dead register values at the entry than the exit of the block, so most buffer maintenance instructions are issued in the prologue.

In the epilogue, the size of the buffer is compared to a threshold value. This threshold value is smaller than the real size of the buffer. The difference should be bigger than the maximum number of instrumentation points of any block in the given application. Function calls are treated as an end to a block. So the buffer is checked only in the epilogue and there is no possibility of buffer overflow. If the stored values are less than the threshold, the epilogue restores regbase to its original value saved in the prologue. Otherwise it jumps to a code sequence which stores all volatile registers and other system status, calls the user provided analysis routine and restores the register values after it returns. This sequence has many instructions and necessitates a control transfer. However it is visited only when the buffer is nearly full. Currently the threshold is set to 10,000 items so this execution path is visited infrequently.

For the common case in which buffer size is less than the threshold and one can find an unused register for regbase, only ten instructions are required in the prologue and five in the epilogue plus one instruction for each instrumentation
point. For example one only needs to insert less than thirty static instructions to collect all the memory traces in the most important of block of CG, which accounts for more than 60% total memory references of the whole program. This is significantly cheaper than making the user maintain the buffer and insert function calls at each instrumentation point. In addition, there are no control interruptions in the middle of the block. Because no original instruction ever touches regbase, there is also no data dependency caused by the inserted instructions. One can take full advantage of the instruction level parallelism mechanisms of Power4. ALITER does issue one conditional branch instruction in the epilogue. However the direction of the branch is very friendly to hardware-based prediction. Additionally, ALITER uses extended branch mnemonics to provide a direction hint for Power4’s branch processing unit.

In the rare case that all the GPRS are touched in the block, ALITER needs four instructions to store register values to the buffer. These are responsible for saving the value of a temporary register, loading the position of the buffer where a value should be written to, storing the effective address to the buffer and at last restoring the value of the temporary register. In the prologue, the position of the buffer is also calculated using a temporary register. The calculated start point is saved in a pre-defined location so it can be loaded later to store the effective addresses. The epilogue is similar to the common case. Of over a thousand basic blocks in the selected benchmarks, less than ten of them touched all GPRs and require special treatment. In fact those blocks are part of XLC system routines.

For certain memory instructions, the static offset information is not available from the assembly code. On Power4, such memory instructions are accessed using register RTOC, a special register which points to the table of contents. It is the linker that finally determines the location of these items in the table and puts the correct offset in these instructions. For such memory instructions one can’t export the offset, since it is not known yet. It is necessary to insert more instructions which first calculate the effective address and then store it to the buffer.
Although ALITER inserts far fewer instructions into the original code compared to other instrumentation tools, it can still expand code size significantly. On Power4 systems, the distance of a conditional branch is limited by 14-bit offset. Although ALITER uses labels in the assembly code and relies on the assembler to calculate the exact distances, the assembly could have trouble encoding the distance in the instrumented code into 14 bits. If that happens, ALITER needs to switch the branch condition and change the branch target. First ALITER creates a new label for the fall through block if it does not have one already. The modified branch is now pointing to the fall through block using its label. A jump instruction is issued between the block branch and its original fall through instructions. This jump branch, which has 24 bits for offset, now points to the original target. ALITER parses the instrumented code a second time after all the instrumented codes have been issued. During the second parse, it calculates the distance for each conditional branch instruction. If the distance is too far, the branch is modified as described above.

3.3.1 User’s Responsibilities

ALITER users are not required to parse the program structure to determine where to insert snippets so that full event traces can be collected (as for example to determine the minimum number of instrumentation points required to capture a complete address stream efficiently). These details are handled by ALITER. Users are only responsible for providing analysis routines that work on an array of events (such as addresses). Because the event buffer is maintained by the instrumentation tool and analyzed by the users analysis routines, it does require mutual understanding of what is stored in the buffer. Although the users do not have to know the exact layout of the buffer, they do have to know whether the buffer holds path traces or memory address traces. How to expressively enable users to specify interesting events is part of future research.
3.3.2 An Example of an Instrumented Block

Figure 3.4 shows the instrumented code of the most important block in CG. In order to help understand the code segment, some instructions are briefly explained here. The details of IBM Power4 ISA is beyond the scope of this thesis and can be found in [88]. Depending on the sizes of the operands, load instructions can have many forms including “l”, “lwz”, and “ldf” etc. “rlwinm” is one shift instruction used in this code segment for calculating the address in the event buffer. It essentially multiplies the buffer size by four, which is the size of each element in the buffer. “bc+” is a form of conditional branch instruction, indicating the branch is very likely to be taken. “bl” is the call instruction on POWER4. There are also a pair of instructions dealing with Conditional Register, which is the special register used to determine branch directions. “mfcfr” which stands for “Move From Conditional Register” copies the content of Conditional Register into the provided general purpose register. “mtcrf” is used in the code segment to overwrite (restore) the content of Conditional Register. All the other instructions are self-explanatory.

The instrumented code has three parts. The first shaded section is the prologue for the basic block whose blockid is 382. ALITER identifies register $r24 and register $r25 have dead values\(^1\). Register $r3 is identified as the base register for this block because it is not used in the original block. The instrumented instructions will use $r3 to access the buffer.

The prologue first moves the value of Conditional Register, which could be changed by inserted comparison instructions, to register $r25. $r25 is identified as one register with dead value therefore it is not required to be saved. Then the value of the base register ($r3) is saved to a special memory location. The next several instructions load the buffer size and compare it to 9000, which is the chosen threshold. If the buffer size is more than the threshold, the prologue calls ProcessBuffer(). The instructions for saving and restoring volatile and special

\(^1\)The instruction which overwrites $r24 is not shown in the figure
registers are omitted in the figure. Most of the time the execution will jump to label \_L3270\_mid. In the rest of the prologue, conditional register is first restored. The buffer size is increased by 14 which is the total number of values will be stored from this block. Then the prologue write the blockid (382) to the buffer. Notice the start address of the buffer is only calculated once in the prologue. It only takes one store instruction to write a value to the buffer.

Only two original instructions are shown in the example.

\begin{verbatim}
lwz r25, 36(r28)
lwz r12, 36(r14)
\end{verbatim}

Both of them are memory instructions and selected to be instrumented. For each of them, there is one store instruction inserted to record the dynamic value of register($r28$ and $r14$ respectively).

Finally in the epilogue, the value of base register $r3$ is restored.

### 3.4 Experiments and Results

Several benchmarks from the NAS Parallel Benchmark Suite [10] were selected to collect their memory traces with three generic binary instrumentation tools. All the memory instructions in the binaries are instrumented with a one-line function, which writes the address into a fixed location and returns. These experiments were designed to measure the cost of collecting the traces. In these experiments, the traces are collected and discarded immediately. These experiments gave a lower bound estimation of slowdown when memory instructions are instrumented. Table 3.1 list overhead of this experiment using three tools. The ATOM data was collected on Lemieux, an Alpha 21264 system at the Pittsburgh Supercomputer Center. The PIN data was collected on Itanium2 nodes belonging to the Teragrid, and located at the National Center for Supercomputer Applications (NCSA), and the Dyninst data was collected on Cheetah, an IBM Power4 at Oak Ridge National Laboratory. We could not finish tracing LU and SP on Chee-
tah using Dyninst in five and a half hours, so the slowdown is under-estimated using 5.5 hours.

Table 3.1: Slowdown of Acquiring Addresses Using Different Tools

<table>
<thead>
<tr>
<th>Application</th>
<th>ATOM</th>
<th>PIN</th>
<th>DyninstAPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG.A.4</td>
<td>53.5</td>
<td>86.6</td>
<td>878.6</td>
</tr>
<tr>
<td>FT.A.4</td>
<td>35.1</td>
<td>51.8</td>
<td>1003.7</td>
</tr>
<tr>
<td>MG.A.4</td>
<td>40.6</td>
<td>57.5</td>
<td>932.5</td>
</tr>
<tr>
<td>LU.A.4</td>
<td>53.0</td>
<td>66.5</td>
<td>&gt;&gt;301.4</td>
</tr>
<tr>
<td>SP.A.4</td>
<td>31.4</td>
<td>47.7</td>
<td>&gt;&gt;203.66</td>
</tr>
</tbody>
</table>

Table 3.2: Slowdown of Acquiring Addresses

ALITER vs. ATOM with delayed instrumentation

<table>
<thead>
<tr>
<th>Application</th>
<th>ALITER</th>
<th>ATOM (delayed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG.A.4</td>
<td>1.7</td>
<td>6.4</td>
</tr>
<tr>
<td>FT.A.4</td>
<td>1.9</td>
<td>2.0</td>
</tr>
<tr>
<td>MG.A.4</td>
<td>1.9</td>
<td>2.5</td>
</tr>
<tr>
<td>LU.A.4</td>
<td>1.2</td>
<td>1.6</td>
</tr>
<tr>
<td>SP.A.4</td>
<td>1.1</td>
<td>2.5</td>
</tr>
</tbody>
</table>

The same benchmarks were then instrumented, compiled with the same flags, using ALITER. This data was collected on Cheetah. Table 3.2 shows the slowdown of collecting memory addresses using ALITER. As can be seen, the benefit of inlining buffer management is tremendous. Table 3.2 also compares the overhead to the best results measured from ATOM using static analysis and delayed instrumentation techniques described in previous chapter. With delayed instrumentation, the most visited blocks in the selected benchmarks, except for CG, only require one instrumentation point to store all the necessary register values per block. Considering ATOM is known to generate a quality instrumented binary, this slowdown in the second column of Table 3.2 is arguably the best results one can get from any generic, synchronous binary instrumentation tools. The overhead of ALITER is still better. In addition, users are freed from the tedious and error prone job of static analysis of the instructions in the binary to determine a minimal set of instrumentation points as well as managing the buffer.
This chapter, in part, is a reprint of the material as it appears in “ALITER: An Asynchronous Lightweight Instrumentation Tool for Event Recording”, by Xiaofeng Gao, Beth Simon and Allan Snavely, published in the proceedings of the 2005 Workshop on Binary Instrumentation and Applications (WBIA05) Sept.05 St. Louis [29]. The dissertation author was the primary researcher and author; the co-authors listed on these publications directed, supervised, or helped the research which forms the basis for these chapters.
L3270:
### prologue for Block 382
mfcr r25  #save Conditional Register
st r3,regbase{TD}(RTOC)
st r25,crreg{TD}(RTOC)
l r24,totalmem{TD}(RTOC)
l r3, T.XXAB.addrbuff(RTOC)
rlwinm r25,r24,2,0,29
cmpi 0,r24,9000  #check buffer size
add r3,r25,r3
bc+ BO_IF_NOT,CR0_GT,__L3270_mid
        < save all volatile registers >
bl .processbuff{PR}  #call ProcessBuffer()
nop
        <restore all volatile registers>
__L3270_mid:
l r25,crreg{TD}(RTOC)
addi r24,r24,14  #total 14 values will be written
mtcrf 255,r25  #restore Conditional Register
addi r25,r0,382  #create blockid
st r24,totalmem{TD}(RTOC)
st r25,0(r3)
### prologue ends here
st r28,4(r3)  #instrumented id:0
lwz r25,36(r28)
st r14,8(r3)  #instrumented id:1
lwz r12,36(r14)

......
### postlogue starts here
l r3,regbase{TD}(RTOC)
### postlogue done

Figure 3.4: The Most Visited Basic Block of CG.S Instrumented Using ALITER
Prologue and Epilogue are marked in the shaded area, only one store instruction is needed to store a register value. All the address calculation is done in the prologue. r3 is selected as the base register in this block
Chapter 4

Reducing Space Cost of Memory Tracing

4.1 Path Grammar Guided Trace Compression

4.1.1 Motivation

Besides the tremendous slowdown one will experience with binary instrumentation tools and processing addresses on-the-fly, such tracing schemes require one rerun the instrumented binary if anything changes in the parameters simulated. During the DOD HPCMO TI05 performance modeling and prediction effort, cache configurations were constantly added or modified, therefore applications had to be instrumented with the new configurations and the whole instrumented version executed over and over again. Even if the users are only interested on the data from one single processor, the whole instrumented parallel applications will have to be traced as a whole. These applications which will run on hundreds of processors for several hours generally will have to wait in the queue for weeks before they can even get started. Not only that, it is also very wasteful of the CPU resources as we have shown that costs of binary instrumentation can account for a significant portion of the overall costs.

One would like to have a physical trace on the hard drive and have an
offline simulation, but unfortunately, even when this process is faster with techniques and tools described in previous chapters, the resulting traces are huge and unwieldy. Just for example, a full uncompressed trace of the effective addresses touched by the NAS Parallel Benchmark [10] BT Class A run on 4 processors takes more than one quarter of a terabyte to store. It should be clear the average user would run out of disk space for acquiring memory traces of full-scale parallel applications. Various compression schemes have been proposed to reduce disk storage requirements. These will be further examined below but they all have a fundamental limitation; these compression methods may or may not result in much compression; it depends on the patterns of events in the raw trace and the compression scheme's ability to recognize and summarize those patterns.

A very common and fast compression scheme, gzip [35, 103], is based on variable-to-fixed length coding. Many reasons can make it less efficient in compressing traces, especially address traces. Methods that are more sophisticated are notoriously slow. For example, Sequitur [63], a widely used compression scheme for architectural research, takes 20 hours (!) to compress just the control flow trace of BT Class A.

Path Grammar Guided Trace Compression (PGGTC) is introduced in this chapter to make event traces storable in reasonable-sized files without requiring potentially huge (unknowable in advance) amounts of scratch disk. This technique keeps control flow information around during the dynamic step and uses it to compress event traces on-the-fly. Such compressed control flow trace files are called Structured Path Histories (SPHs). Combined with gzip, PGGTC on average is over 40 times faster than methods such as Sequitur and requires no scratch disk. Its compression ratios are not quite as high as Sequitur but much better than directly using gzip and can do better than Sequitur after post-analysis, thus hitting a sweet spot for time and space required to capture and store event traces. Most importantly, because control flow information is preserved and stored with the event trace as part of the compression scheme, one can flexibly decompress
only events pertaining to code sections of interest. This flexibility makes dealing with large event traces more wieldy; it also enables some advanced uses such as synthetically generating an event trace for the program run on an input different from that with which the original trace was gathered.

This chapter also deals with the following problem: some kinds of program events defeat even sophisticated compression. A good example is a sequence of truly random effective memory accesses. If there is no detectable pattern to these addresses there can be, by definition, no summary to fully characterize them. Such events are the bane of the tracer as they cost an inordinate amount of time to gather and cannot be compressed. By contrast, highly regular event patterns can often be determined via low-overhead static analysis before tracing and then just a few details, such as number of times the pattern is encountered in branch/loop trip counts etc. can be filled in by lightweight tracing. But potentially non-patterned events, such as the aforementioned random memory addresses that might result from a series of pointer references, need to be monitored dynamically during tracing and may result in sequences that are hard or impossible to compress. This thesis approaches this problem with a variety of techniques: 1) applying static analysis extensively and focusing on dynamic tracing on only those events that cannot be determined by static analysis 2) using sampling to statistically characterize potentially random events without storing all of them. 3) extending this idea to Trace Approximation (TA).

TA leverages the following idea: many times, for trace-driven simulation and validation of benefit of architectural features, one would be satisfied with a ‘performance similar’ trace. That is, from the performance standpoint, it may be that any sequence of random memory accesses with the same statistical distribution would perform the same against the architectural feature being evaluated. Therefore one can replace traces of random addresses with other (more compressable) random addresses. Using TA technique one can obtain a compact event trace from a program that, when replayed against a simulator, will reproduce the statis-
tical behavior of the events (statistics defined by the user) of the original program without guaranteeing exact duplication of particular events in sequence (such as exact sequence of random memory addresses touched).

### 4.1.2 Limitations of Current Compression Schemes

Trace files can be compressed to reduce their storage size. The most common way is to just use gzip. gzip commonly results in about two order-of-magnitude reduction in file size. There have been many techniques developed to compress control flow traces and associated effective address traces to achieve, commonly, as high as 6 orders of magnitude reduction in file size. These include Sequitur, VPC [22] and SIGMA [24]. However they are much slower than gzip. They also can achieve high compression ratios only when there are detectable patterns in the address sequence. Uncertainty as to compression ratio is very undesirable because, for a previously unstudied application, the user has no clue of disk resource requirements for saving the trace or even feasibility of saving the trace. One might need a few kilobytes of storage or several terabytes.

![Figure 4.1: A Control Flow Example](image)

Let us examine some scenarios where Sequitur does poorly. The Sequitur algorithm tries to build a context free grammar from a sequence (for example of blockids) and uses this grammar to compress the trace. The algorithm goes through
the elements in the sequence and creates rules based on observed repetitions. If there is a sub-sequence that has been repeated more than a threshold, the algorithm creates a reduction rule and replaces future appearances of the sub-sequence with the rule. The process repeats recursively until there are no more rules to create. For example consider an inner loop shown in Figure 4.1. Suppose in block C, function 2 is called and it always returns to the caller. For the following trace:

XACFGGHHIBDABDACFGHIBDABDACFGGGHHHI
BDABDACFGHHHHIBDABDE

the Sequitur algorithm has created the following rules:

0 -> X A 1 2 3 3 G G 2 H 3 H H 4 5 E 
1 -> C F G
2 -> G H
3 -> 4 6 1
4 -> H I 6
5 -> B D
6 -> 5 A

The trace can be recovered by replacing non-terminals (numeric rules) in rule 0. Examination shows that most of the rules Sequitur has “discovered” are just recaps of what is obvious in the control flow graph.

Further examination highlights limitations of Sequitur and others stream compression techniques, which stem from the fact they treat program traces as streams of completely unknown events without regard to the control flow. In addition, the hierarchical nature of the program trace (i.e. nested loops and function calls) is not considered by algorithms such as Sequitur. The detected patterns may be mixed up between functions and loops and branches etc. Many patterns indeed will not be detected because they are obscured by some non-patterned elements that happens in the middle from a function call and the like. In the example above
Sequitur fails to reveal a much more interesting pattern in the loop: two paths are taken alternatively ACBD, ABD, ACBD, ABD.

Algorithms such as Sequitur, generally are troubled by non-patterned elements in the sequence. Too many rules and patterns are created to attempt to summarize things that may not have a pattern at all. These rules and patterns increase the sizes of look up tables and make the compression process much slower than it would otherwise be. For example, as will be seen in the results section, Sequitur has problems with NAS Parallel EP, which has a major loop which is basically random. Sequitur could not finish compressing the 944MB control flow trace in 70 hours.

4.1.3 Creating Path Grammar from Static Analysis

Building a context free grammar that basically captures (at least some aspects of) the CFG as Sequitur does, is redundant for the purposes of compressing an application trace. In fact the control flow is determined by the static control flow graph, which is already a grammar and is already known at compile time and can be used directly for compression. The binary is first parsed to build a CFG by identifying basic blocks, edges and natural loops. Then basic blocks are assigned to structures. A structure is either a natural loop, or a function. Each basic block (single entry, single exit instruction sequence) is only assigned to the innermost structure that contains this block. Each structure has its own path grammar; the grammar is built from the sub-graph that only contains the basic blocks that belong to this structure. These grammars will be used to encode paths taken within the structure during execution.

A path value represents a particular path through a structure and is encoded as a bit string. For each branch in a structure, a unique bit is reserved. Taking one edge will set the bit to zero and the other edge will set the bit to 1. This is a similar approach to that used in DYNAMO [11] and TFP [59] but this thesis uses one bit per branch instead of one bit per block to reduce the number
of bits required. Now for most cases, comparing two path values is effectively comparing two unsigned 32-bit integers as most structures do not contain more than 32 branches. For all the NPB benchmarks, only 14 structures required bit string longer than 32 bits. For the rare structures that require more than 32 bits to present all its branches, the bit string is hashed into a 32-bit value that must be looked-up during tracing and compression. In the inner loop in Figure 4.1, there are three branches so three bits are required for representing the paths in this structure. For example the first branch starting from block A is associated with the lowest bit. When edge (A,C) is taken, the bit will be set 1 (the path value would be xx1 depending on the other two branches).

For tracing and compression, edges are are associated with actions based on the grammar, such as to set the bit strings. Unlike other compression schemes, these actions are determined during static analysis and blocks are instrumented accordingly so there is no need to create rules and look up a dictionary to determine what should be done when a particular edge is taken. For example the actions assigned to edge (D,A) include the following three actions. First the current path value is BitOred with its bit mask (000). Secondly (D,A) indicates a completion of current path. The path value is added to a history buffer. Thirdly the path value is reset for the next pass of the loop. The structures of the control flow graph shown in Figure 4.1 are defined as

\[
\begin{align*}
\text{struct0} &= \{X,E\} \\
\text{struct1} &= \{A,C,B,D\} \\
\text{struct2} &= \{G\} \\
\text{struct3} &= \{H\} \\
\text{struct4} &= \{F,I\}
\end{align*}
\]

Natural loops sharing the same loop head are regarded as one structure. For structure 1, static analysis will determine the bit mask for each branch and the actions for each edge. Not all edges need actions. Actions that BitOr 0 can be omitted. Here now are the actions associated with the edges of structure 1:
(X,A)=>{PushStack()
  struct1.NewInstance()}
(A,C)=>{struct1.pathval.BitOr(001)
  RegisterCallSite()}
(B,A)=>{struct1.pathval.BitOr(010),
  struct1.AddPathValToHistory(),
  struct1.ResetPathVal()}
(D,A)=>{struct1.AddPathValToHistory(),
  struct1.ResetPathVal()}
(D,E)=>{struct1.pathval.BitOr(100),
  struct1.AddPathValToHistory(),
  struct1.CompleteInstance(),
  PopStack()}

4.1.4 PGGTC Online Procession

PGGTC compresses the traces for each structure individually and stores them independently. The compressed control flow trace files are called Structured Path Histories (SPHs). Although SPHs also deal with whole program path histories, it differs from Whole Program Path (WPP) [47] in three major aspects. First the path histories are recognized, processed and saved individually and differently based on structures. Post-processing can work on each structure’s history without decompressing the whole trace. It is not only easier and faster to work on parts of a trace like this, patterns are easier to identify when elements from other structures are not intermixed. Secondly the paths and actions are defined during static analysis. The online analysis is considerably lightweight compared to other approaches which have to build the dictionary during the online analysis. Thirdly the grammar is the same for traces from different runs of the same application.

Once the structures are defined and actions have been associated with edges, blocks are instrumented accordingly to implement the defined actions. On-
line analysis will capture the blockids, use one look back entry to determine the edges and follow the actions associated with the edges to compress the trace. When an edge completes a path through a structure, for example either a back edge such as (D,A) or an exit edge such as (D,E), then the bit string representing the completed path value is added to a history buffer attached to that structure. A structure’s history buffer is organized as a pair of values consisting of a path value (bit string) and a repetition integer. If the bit string is the same as the previous one in the buffer, the repetition integer is incremented, otherwise a new pair of values is added to the buffer. When a loop instance completes, a special mark is added to the buffer to indicate the completion of that instance of the loop so that future instances of the loop will start from a new pair. When the buffer is full, one simply uses any lossless compression scheme, such as the utility functions provided in zlib [104] (gzip) to compress it and append it to the structure’s SPH. Such a file is then a history of the structure recording all the paths taken in every dynamic instance of it.

Recursion must be dealt with. An execution stack is used in the implementation to overcome the problems caused by multiple instances of the same structure. Every time a path is completed, it is added to the history of the most recent instance of the structure. Keeping the execution stack increases overhead significantly, if recursion is guaranteed not to occur, the implementation could be sped up by using a simple version. The trace shown previously in Section 4.1.2 is represented by the following path histories of 5 structures. As an example, (0:3,1:1.) indicates Path 0 is taken three times and then followed by Path 1 once and the loop instance completes.

\[
\text{struct0: \{(0:1)\}}
\]

\[
\text{struct1: \{(1:1,0:1,1:1,0:1,1:1,0:1,1:1, 4:1.)\}}
\]

\[
\text{struct2: \{(0:1,1:1.) (1:1.) (0:3,1:1.) (0:1.)\}}
\]
struct3: {(0:1,1:1.) (1:1.) (0:2,1:1.) (0:2,1:1.)}

struct4: {(0:4)}

The targets of certain instructions are unknown during static analysis. They can also have more than 2 followers during execution. Such instructions include jump and call instructions using registers for addressing. Blocks that end with such instructions are called call sites. The online analysis keeps a buffer of followers for each of these call sites. Normally a call site has only one or a very limited number of followers, so gzip can compress the follower buffer with high compression rate.

When one wants to examine or replay the trace, path values are replaced with sequences of blocks. For example, the decompression process will replace path0 of structure 1 with ABDA. When a block in the sequence is a call site (for example block C), its next follower and the follower’s instance are filled in after the call site. Each structure maintains its own pointer to the next instance from its history file. Here are some path values decompressed to sequences of blocks for structure 1:

path0(val=000) => ABDA
path1(val=001) => AC(...)BDA
path2(val=010) => ABA
path3(val=011) => AC(...)BA
path4(val=100) => ABDE
path5(val=101) => AC(...)BDE

SPH is a lossless representation of the control flow trace. The correctness can be validated by comparing the original stream of blockids to the decompressed ones. PGGTC achieves high compression ratios for its SPHs by transforming a sequence of events into a representation that is friendly to low-level compression schemes. It does this by keeping separate histories for each structure and not allowing them to get intermixed. Because each structure has its own bit string, the possible values
of the bit string depends on how many decision points (branches) there are in the structure. For the most important inner loops of scientific applications, there are not too many branches. They may be typical for example of G and H in Figure 4.1 which have only have two possible path values. A limited range of path values enables gzip to achieve very high compression ratio as will be seen in the results (next).

4.1.5 Results

The NAS Parallel Benchmarks were instrumented, and their control flow traced, using ATOM [85] on an Alpha-based supercomputer at the Pittsburgh Supercomputing Center. Each basic block in each application was instrumented to write out the blockid. Although it is not necessary to save the trace first onto disk with PGGTC, this is done in the following set of experiments just to compare the speed of PGGTC compression against other methods. The compression experiments were all done on an Intel workstation with 1GB memory and a 2.8GHz P4 processor. The results are shown in Table 4.1 and Table 4.2.

PGGTC+gzip, which uses gzip to compress path histories, is on average 24 times slower than using gzip, but the sizes of compressed files are on average over 300 times smaller. Sequitur source code was downloaded from http://sequitur.info/ and applied to the uncompressed traces. Sometimes Sequitur yields compressed files as much as five times smaller than PGGTC with gzip, sometimes the compressed files are actually slightly larger, but the compression takes on average 40 times longer than PGGTC+gzip; this does not seem like a reasonable time/space tradeoff. In addition, Sequitur must put full trace on the scratch disk before compression and apparently has trouble handling big traces with highly random contents (such as EP trace).

Lastly, PGGTC was used with Sequitur as the underlying low-level algorithm for compressing path histories (PGGTC+ Sequitur). Because Sequitur source code does not support accumulative compression as gzip does, it is applied
on decompressed SPH files from PGGTC+gzip. The time in the last column includes the extra time for PGGTC+gzip. PGGTC + Sequitur compresses traces to be more than two times smaller, and does it 14 times faster, than using Sequitur alone. The last row lists the average gain in compression ratio over gzip and the slowdown over using gzip.

Table 4.1: Compression Ratios and Compressing Time: Part 1

<table>
<thead>
<tr>
<th>Application</th>
<th>uncompressed</th>
<th>gzip</th>
<th>PGGTC+ gzip</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT.A.4</td>
<td>3.26GB</td>
<td>15.5MB (77.9sec)</td>
<td>23.9KB (1384 sec)</td>
</tr>
<tr>
<td>CG.A.4</td>
<td>253.3MB</td>
<td>6.03MB (12.9sec)</td>
<td>2.39MB (156 sec)</td>
</tr>
<tr>
<td>EP.A.4</td>
<td>944.5MB</td>
<td>15.2MB (30.7sec)</td>
<td>9.1MB (566 sec)</td>
</tr>
<tr>
<td>FT.A.4</td>
<td>558.9MB</td>
<td>3.6MB (14.0sec)</td>
<td>24.4KB (322 sec)</td>
</tr>
<tr>
<td>IS.A.4</td>
<td>343.8MB</td>
<td>506.1KB (6.9sec)</td>
<td>1.9KB (399 sec)</td>
</tr>
<tr>
<td>LU.A.4</td>
<td>873.8MB</td>
<td>5.2MB (18.9sec)</td>
<td>17.1KB (466 sec)</td>
</tr>
<tr>
<td>MG.A.4</td>
<td>431.8MB</td>
<td>14.9MB (10.2sec)</td>
<td>21.8KB (192 sec)</td>
</tr>
<tr>
<td>SP.A.4</td>
<td>5.04GB</td>
<td>26.9MB (121.6sec)</td>
<td>49.6KB (2616 sec)</td>
</tr>
<tr>
<td>Average</td>
<td>1X(1X)</td>
<td>331.6X(24.3X)</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: Compression Ratios and Compressing Time: Part 2

The last row lists the average gain in compression ratio over gzip and the slowdown over using gzip, excluding EP.

<table>
<thead>
<tr>
<th>Application</th>
<th>uncompressed</th>
<th>sequitur</th>
<th>PGGTC + sequitur</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT.A.4</td>
<td>3.26GB</td>
<td>14.0KB (19.9hrs)</td>
<td>5.81KB (2936sec)</td>
</tr>
<tr>
<td>CG.A.4</td>
<td>253.3MB</td>
<td>349.2KB (1.5hrs)</td>
<td>241.2KB (551sec)</td>
</tr>
<tr>
<td>EP.A.4</td>
<td>944.5MB</td>
<td>N/A (&gt;70hrs)</td>
<td>N/A (&gt;70hrs)</td>
</tr>
<tr>
<td>FT.A.4</td>
<td>558.9MB</td>
<td>4.4KB (3.7hrs)</td>
<td>2.82KB (1007sec)</td>
</tr>
<tr>
<td>IS.A.4</td>
<td>343.8MB</td>
<td>2.2KB (1.8hrs)</td>
<td>805B (1021sec)</td>
</tr>
<tr>
<td>LU.A.4</td>
<td>873.8MB</td>
<td>6.6KB (5.4hrs)</td>
<td>4.25KB (1910 sec)</td>
</tr>
<tr>
<td>MG.A.4</td>
<td>431.8MB</td>
<td>34.3KB (2.5hrs)</td>
<td>10.62KB (353 sec)</td>
</tr>
<tr>
<td>SP.A.4</td>
<td>5.04GB</td>
<td>28.5KB (31.5hrs)</td>
<td>8.27KB (3.3hrs)</td>
</tr>
<tr>
<td>Average</td>
<td>633.9X(868.1X)</td>
<td>1530.5X(76.3X)</td>
<td></td>
</tr>
</tbody>
</table>

4.1.6 Trade-off between Time and Space

Although PGGTC+gzip does not achieve compression ratios as high as Sequitur, the compressed trace sizes are acceptable for most users. Besides being
40 times faster, PGGTC+gzip does not require scratch disk; the traces can be compressed on-the-fly. This thesis argues that PGGTC+gzip hits the sweet spot in compressing control flow traces. Figure 4.2 shows the time and space costs of different approaches of handling the traces. The dots from the left to right are, uncompressed, compressed using gzip, compressed using PGGTC with gzip, compressed using PGGTC with Sequitur, compressed with sequitur.

![Figure 4.2: Time and Space Costs of Compressing Control Flow Traces](image)

The first dot of each application is not compressed. The second one is compressed using gzip, then PGGTC +gzip, PGGTC+Sequitur and finally Sequitur.

### 4.2 Advanced Features of Structured Path Histories

The inherent hierarchical structure of a program preserved in SPH files, besides enabling good compression, enables study of the dynamic behaviors of ap-
lications. Unlike the other path-based traces such as WPP, which generates one, possibly huge, monolithic file which is difficult to analyze and unwieldy, each SPH file keeps part of the trace according to its natural structures and stores the histories of each structure individually. Each file can be decompressed and analyzed independently. And these smaller files are much easier to handle. In addition, SPH files encapsulate loop instance markers in the compressed traces. This enables one to perform extra analysis on higher levels such as detecting the repetitive patterns at the instance level. Currently, in order to reduce time overhead, PGGTC only detects immediate repetitive patterns at path level during online analysis. Post analysis can be used however to detect higher level patterns. For example, a repeated pattern in loop instances such as \((1:1,0:1,1:1,0:1,1:0,1,1:1,4:1)\) can be detected during post analysis. In addition, SPH enables one to detect the differences between two different runs of the same application, either for performance modeling or debugging. One major problem of diffing two traces is how to determine the synchronization points. With SPH, the traces can be compared at the structural level. It is easy then to tell which structure’s behaviors are exactly the same and which are different during different runs (as with different inputs for example). The instance marker in the history can be used to align the comparison at the instance level. It is possible to pinpoint the exact instance at which two traces become different. Rules to encode paths are derived from static binary, so bit strings can be compared directly from different traces of the same application.

As an example of this diffing capability, this thesis tested the code shown in Figure 2 with four sets of inputs where an input is a pair of loopcount and runlen: \((18, 3)\) \((20, 4)\) \((33, 7)\) \((45,5)\). A script is used to compare the traces. The script calls the UNIX diff command to compare the structure histories with the same file names in different directories. Only the files that are different are printed. In this case, the history files of structure 21 (the inner loop of AlterPath()) are different when different inputs are provided.
The high level point is that the compression scheme enables tractable comparison of traces at the structural level and can be used for simple, intuitive, comparisons of traces such as differencing.
4.2.1 SPH Alternation and Synthesis

If the kernel of an important application is not stable and behavior is dramatically different depending on input, multiple traces should be used to feed the simulator so that the simulated results will be more representative of a real workload. Collecting, compressing, saving and replaying multiple traces of every possible input, even using PGGTC, be too time-consuming to be practical. To address this, SPHs can be artificially generated to reflect what would be the resulting traces from the program run with different inputs, and plug into existing SPHs collected from a real run. For example, the path histories of the inner loop of function AlterPath() in Figure 4.3 is not stable; histories are determined by the inputs. Compiler optimization inserted an extra branch in the loop. If the second parameter is $2^k - 1$, the first branch will always be taken so the possible path values are 7 (111) and 5 (101) for paths ending with the back-edge, and 3 (011) and 1 (001) for the paths that exit the loop. Otherwise the possible path values are 6(110), 4(100), 2 (010) and 0(000). Path 0 and 2 (or 1 and 3) can only be the last path because they both indicate the back-edge is not taken so they will complete the loop instance. The repetition of path 4 (or 5) will equal the second parameter. The length of the trace is determined by the first parameter. If such input-dependent behaviors are already known, one can simply generate the trace for just one set of parameters, say (45, 8) with the resulting SPH:

$6:1,4:8,6:1,4:8,6:1,4:8,6:1,4:8,6:1,$
$4:7,0:1.$

and then the trace for another set of parameters, say (108,15), will trivially be:

$7:1,5:15,7:1,5:15,7:1,5:15,7:1,5:15,7:1,5:15,$
$7:1,5:15,7:1,5:15,7:1,5:10,1:1.$

Synthetic SPH files can be used together seamlessly with SPHs of other structures that are acquired from actual instrumented runs. A generator can be used to systematically generate SPHs to cover all the interesting cases without running the
instrumented binary to collect them. Another extreme example, is if it is known beforehand that paths of certain structure will be random (as is the case for EP), there is no need to waste time and space to collect and compress the traces from such a structure. The decompressor of that particular structure can be modified to issue random valid path histories instead of regenerating block sequences from recorded histories.

4.3 Path Grammar Guided Trace Approximation

So far in the discussion this thesis has focused just on control flow traces. Control flow traces are relatively small to begin with; address traces are much bigger and more difficult to compress. Yet address traces, that is the history of memory locations touched by an application, are among the most important event sequences for simulating how a program interacts with a machine’s memory hierarchy [18]. Unlike control flow traces, repetition in effective addresses is less likely so repetition based compression schemes do not work well directly. Effective addresses are typically compressed using offsets, similar to Mache [73] and PDATS [41]. However even the most sophisticated compressing schemes do poorly when the address sequence is mostly random and resulting compressed trace is almost the same size as the uncompressed one. Such sequences are unfortunately common in scientific applications that deal with sparse data, so it is impractical to have any lossless compressed effective address traces for these applications. However, for many performance related studies that consume addresses traces, such as cache simulation, there is no need to keep the exact addresses in order to get the same or very similar simulation results. An address sequence approximately similar to the application’s true address sequence, a sequence that can be represented in a compact manner, may be just as useful as long as it behaves in a manner similar to the original. In this section this thesis discusses two approaches to collect information from traces for generating approximate ones 1) selective dumping and
2) summarized memory signatures. The SPHs are used as the backbone of the execution and approximated addresses are generated by “decorating” SPHs with additional address information. To validate accuracy this thesis compares cache simulations against full address sequences and selective or approximated sequences.

4.3.1 Static Data Flow Analysis

For memory tracing the first strategy is to avoid it as much as possible. Dynamic memory tracing is notoriously slow. One should attempt to ascertain as much as one can about the program’s address sequence via static analysis (much faster) and identify the minimum set of load and store instructions in the binary that must be instrumented and traced to obtain a complete address trace. This thesis leverages the fact many loads and stores have some constant static relationship and one needs only know one of the set to determine the others. In Chapter 2 and Chapter 3, this thesis showed that if every memory instructions is instrumented, then typically 10-fold slowdown over un-instrumented execution is a lower bound. The slowdown is typically much higher if there is any actual dynamic analysis. Such high overhead makes it infeasible to completely trace larger applications online. Fortunately, static data-flow analysis can be used to group memory instructions into static sets if the register values used by the instructions are guaranteed to be the same or to have some fixed, predetermined offset. Only one memory instruction is needed to be instrumented from each group, and the

<table>
<thead>
<tr>
<th>App.</th>
<th>Inst.Level</th>
<th>Block Level</th>
<th>Mem Inst.</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT.A.4</td>
<td>300</td>
<td>1105</td>
<td>18714</td>
</tr>
<tr>
<td>CG.A.4</td>
<td>141</td>
<td>579</td>
<td>3489</td>
</tr>
<tr>
<td>EP.A.4</td>
<td>45</td>
<td>184</td>
<td>828</td>
</tr>
<tr>
<td>FT.A.4</td>
<td>225</td>
<td>633</td>
<td>4158</td>
</tr>
<tr>
<td>IS.A.4</td>
<td>22</td>
<td>155</td>
<td>462</td>
</tr>
<tr>
<td>LU.A.4</td>
<td>249</td>
<td>966</td>
<td>18396</td>
</tr>
<tr>
<td>MG.A.4</td>
<td>439</td>
<td>1124</td>
<td>9158</td>
</tr>
<tr>
<td>SP.A.4</td>
<td>338</td>
<td>1732</td>
<td>34166</td>
</tr>
</tbody>
</table>
effective addresses of the others can be calculated based on the known offsets. On average, over 90% of the load/store candidates for instrumentation can be eliminated in this way. For the two schemes described in the next two sections, on-line tracing is done only on the loads and stores determined to be in the minimal set required to determine all address values. Table 4.3 shows the total number of instrumentation points required to ascertain all memory addresses compared to the total number of memory (load and store instructions) for the NPB. The instrumentation uses the sum of the first two columns where the first column is the total number of load and store instructions instrumented for address tracing, the second is the number of instrumentation points required to do control flow tracing, and the last column is the total number of memory instructions in the program (candidates for memory tracing by a naive scheme).

### 4.3.2 Selective Dumping

Groups are analyzed to determine static stride patterns. Loads and stores without detectable static strides can be sampled to understand how they access the memory. Sampling has already been shown to be effective in that processing only a small fraction of the overall addresses and interpolating the others can generate simulation results with a high degree of verisimilitude to full traces [45, 94]. A contribution of this thesis is to focus sampling on just those instructions with no detectable static stride patterns. For example in the CG kernel, two of the three loads have static stride patterns. If the addresses are sampled equally, only one third of collected addresses are actually necessary. The effective addresses of the other two groups can be exactly regenerated given just their starting address and total number of accesses. Thus selective sampling is only applied to instructions without static patterns (as determined by static analysis). Given the same amount of disk space, Selective Dumping can provide much more address information.

SPH files are used as the backbone to connect all the individual pieces and provide the sequential information required to regenerate the address sequence. In
the simulation phase, simulators are driven by generated approximate addresses. The number of memory instructions and their order, offset, groups on a path are all statically known. Effective addresses are generated and issued according to the execution paths recorded in SPH. If a memory instruction belongs to a group whose static stride is known, the effective address is generated sequentially from the start address of the group (via tracing). For the memory instructions whose groups have been selectively dumped, the effective addresses are pulled from the sampled addresses.

### 4.3.3 Generating Effective Addresses from Memory Signatures

Many scientific applications are programmed in a refining style, which means the program changes behaviors over time. Selective dumping without awareness of phase changes [66, 49] can cause important data points to be missed [50]. PGGTC can detect certain phase changes in control flow and these can be used to enable even more intelligent selective dumping. However, like any other control flow based phase mechanism, the detected phases in control flow are not always correlated to actual changes in the address sequence and vice versa. For example, for a loop that uses an index array to access a sparse matrix, the data access patterns can be completely different for different instances of the loop even if the control flow of two instances of the loop are exactly the same. This happens when the index array are assigned to different regions in the matrix which have different locality characteristics.

As mentioned before, it may not be necessary to record the exact addresses for non-patterned groups. Previous work [30] has shown that even uniformly random addresses [46] generated in the observed range can be used to approximate addresses with fair success. PGGTC can be used to guide how and when memory access signatures could be summarized and stored according to more sophisticated statistical analysis. A memory access signature of a group is a summary of how the effective addresses of the instruction group are distributed
in memory during one instance of the structure. As with selective dumping, this thesis only considers processing addresses of groups without static stride patterns. Different analysis or simulators may have different focus on the properties of the address stream. For analysis that determines hot spots in memory, a histogram analysis of the addresses will be sufficient. This thesis focuses on cache simulation and in what follows the memory signature is a tuple of 11 values. Two values record the region in memory space that are active during the instance of the structure. Eight values are used to record stride distribution indicate how the addresses jump within the region. Every captured address is compared to the previous address of the same group to calculate stride. This stride is then put into one of 8 buckets of strides, covering various stride values from less than 8 bytes, to over 512 bytes. The last value is used to measure the locality score in the short term cache. This short term locality cache is used to identify the spatial locality and reuse among addresses from all groups. A traditional way of finding short term locality is to have a global observation window which holds some previously captured addresses. Each captured address, from any group, is looked up in the window to determine the shortest distance to a previous address. To make this approach effective, the observation window has to be relatively big. Increasing the window size will substantially increase the overhead of on-line processing. To approximate this more quickly, a direct mapped cache is used to detect localities. Every address captured is used to update the cache, but the miss is only counted if the stride is larger than a threshold, for example 64 bytes. Unlike the filter cache used in various filtering based trace reduction schemes, the cache here is not for discarding addresses. It simply is used to measure locality attributes.

The disk space required to save the memory signature is relative small and can be estimated from the SPHs. For a group that is statically classified as strided, one value is required for each structure’s instance. For other group, 11 extra values are needed per instance of the structure.

Approximate effective addresses can be re-generated from the recorded
information as follows: If the group has been statically determined as strided, the addresses from this group are simply generated from the recorded (by tracing) start addresses and using the fixed stride. For such groups, the address sequence generated will be exactly the same as the original. For the groups that are classified as non-patterned from static analysis, the addresses are generated from the signatures and could be different from the original. If all strides of the group are in the same bucket, the stride is the average stride calculated from lowest address, highest address and the counter of the bucket. The approximated address sequence starts from the lowest address and is incremented by the calculated average stride.

For other groups, a random number [53] is used to select the stride based on the histogram of the stride bins. If the chosen stride is larger than the threshold, another random number is used to determine whether this access should be a hit or miss according to the miss rate in the locality cache. If this address is selected to be a hit, the address generation routine will check the cache and make sure the generated address is a hit otherwise it will be regenerated. It is worthwhile to note that although the cache is used only for strides larger than the threshold, all generated addresses, even from groups that are statically determined as strided, update the cache.

4.3.4 Results

Disk Requirement for Trace Approximation

SPHs are required to generate the approximate addresses from selectively dumped addresses or memory signatures. SPH can replay the whole execution from the very beginning or from any given entry point via an extra synchronization file. These synchronization files tell how the instances of nested structures are related. These files can be created from the SPH during post processing or created at the same time as the SPHs are being generated. For selective dumping the experiments only dumped the addresses of selected structure’s groups for the first 5000 dynamic structure instances. Such a small sampling period is sufficient in this case because
the NPB benchmarks are inherently repetitive and have no dramatic phase changes. For real applications, longer and more frequent dumping for the selected structures may be necessary.

Table 4.4: Size of Full trace, Selective Dumped Trace and Memory Signature

<table>
<thead>
<tr>
<th>Application</th>
<th>FULL EA trace</th>
<th>Size of SPH</th>
<th>Selective Dumping</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT.A.4</td>
<td>263.5GB</td>
<td>23.9KB</td>
<td>151.8KB</td>
</tr>
<tr>
<td>CG.A.4</td>
<td>5.4GB</td>
<td>2.3MB</td>
<td>8.59MB</td>
</tr>
<tr>
<td>EP.A.4</td>
<td>7.4GB</td>
<td>9.1MB</td>
<td>25.2KB</td>
</tr>
<tr>
<td>FT.A.4</td>
<td>18.7GB</td>
<td>24.4KB</td>
<td>441.5KB</td>
</tr>
<tr>
<td>IS.A.4</td>
<td>3.1GB</td>
<td>1.9KB</td>
<td>688.4KB</td>
</tr>
<tr>
<td>LU.A.4</td>
<td>158.9GB</td>
<td>17.1KB</td>
<td>105.1KB</td>
</tr>
<tr>
<td>MG.A.4</td>
<td>12.6GB</td>
<td>21.8KB</td>
<td>192.9KB</td>
</tr>
<tr>
<td>SP.A.4</td>
<td>161.2GB</td>
<td>49.6KB</td>
<td>346.7KB</td>
</tr>
</tbody>
</table>

Table 4.5: Size of Memory Signature and SyncInfo

SyncInfo is an extra file used to help extract histories from any given entry point. Otherwise the histories have to be decompressed from the very beginning.

<table>
<thead>
<tr>
<th>Application</th>
<th>Memory Signatures</th>
<th>SyncInfo</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT.A.4</td>
<td>3.6MB</td>
<td>7.6MB</td>
</tr>
<tr>
<td>CG.A.4</td>
<td>1.7MB</td>
<td>9.5MB</td>
</tr>
<tr>
<td>EP.A.4</td>
<td>25.6KB</td>
<td>469B</td>
</tr>
<tr>
<td>FT.A.4</td>
<td>647.5KB</td>
<td>15.5KB</td>
</tr>
<tr>
<td>IS.A.4</td>
<td>8.6KB</td>
<td>1.2KB</td>
</tr>
<tr>
<td>LU.A.4</td>
<td>3.4MB</td>
<td>5.9KB</td>
</tr>
<tr>
<td>MG.A.4</td>
<td>4.7MB</td>
<td>5.9KB</td>
</tr>
<tr>
<td>SP.A.4</td>
<td>9.5MB</td>
<td>12.9KB</td>
</tr>
</tbody>
</table>

Table 4.4 shows the size of a full address trace of each of the NPB in column one, the size of the SPH information in the column labeled Size of SPH (column three), the size of the selective dumped traces. Table 4.5 shows the sizes of memory signatures, and aforementioned synchronization information required to replay the trace from any arbitrary starting point. It can be seen that selective dumps and memory signatures are very small compared to full address traces.
Table 4.6: Average Absolute Errors of Using Approximate Traces

<table>
<thead>
<tr>
<th>Application</th>
<th>Error Using Selective Dumped Traces</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>L1 Error (%)</td>
<td>L2 Error (%)</td>
<td>L3 Error (%)</td>
<td></td>
</tr>
<tr>
<td>BT.A.4</td>
<td>1.60</td>
<td>2.15</td>
<td>0.37</td>
<td></td>
</tr>
<tr>
<td>CG.A.4</td>
<td>1.34</td>
<td>1.36</td>
<td>0.18</td>
<td></td>
</tr>
<tr>
<td>EP.A.4</td>
<td>0.04</td>
<td>0.22</td>
<td>0.16</td>
<td></td>
</tr>
<tr>
<td>FT.A.4</td>
<td>2.78</td>
<td>2.64</td>
<td>0.40</td>
<td></td>
</tr>
<tr>
<td>IS.A.4</td>
<td>0.50</td>
<td>0.62</td>
<td>0.09</td>
<td></td>
</tr>
<tr>
<td>LU.A.4</td>
<td>2.00</td>
<td>1.63</td>
<td>0.18</td>
<td></td>
</tr>
<tr>
<td>MG.A.4</td>
<td>2.34</td>
<td>1.75</td>
<td>0.43</td>
<td></td>
</tr>
<tr>
<td>SP.A.4</td>
<td>1.87</td>
<td>2.62</td>
<td>0.55</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.7: Average Absolute Errors of Using Approximate Traces

<table>
<thead>
<tr>
<th>Application</th>
<th>Error Using Memory Signatures</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>L1 Error (%)</td>
<td>L2 Error (%)</td>
<td>L3 Error (%)</td>
<td></td>
</tr>
<tr>
<td>BT.A.4</td>
<td>2.03</td>
<td>2.00</td>
<td>0.51</td>
<td></td>
</tr>
<tr>
<td>CG.A.4</td>
<td>1.50</td>
<td>1.00</td>
<td>0.16</td>
<td></td>
</tr>
<tr>
<td>EP.A.4</td>
<td>0.11</td>
<td>0.70</td>
<td>0.16</td>
<td></td>
</tr>
<tr>
<td>FT.A.4</td>
<td>2.98</td>
<td>1.65</td>
<td>0.27</td>
<td></td>
</tr>
<tr>
<td>IS.A.4</td>
<td>3.15</td>
<td>2.10</td>
<td>0.23</td>
<td></td>
</tr>
<tr>
<td>LU.A.4</td>
<td>3.45</td>
<td>2.15</td>
<td>0.54</td>
<td></td>
</tr>
<tr>
<td>MG.A.4</td>
<td>3.81</td>
<td>2.95</td>
<td>0.79</td>
<td></td>
</tr>
<tr>
<td>SP.A.4</td>
<td>4.25</td>
<td>3.02</td>
<td>0.86</td>
<td></td>
</tr>
</tbody>
</table>

Validation of Approximate Traces

Table 4.6 compares simulation results using generated approximate traces from selectively dumped addresses to simulation results using complete traced addresses. Table 4.7 compares simulations results using generated approximated traces from memory signatures to simulation results using complete traced addresses. The simulator simulates 25 cache configurations listed in Table 6.1 and Table 6.2. It generates cache hit rates for each basic block and for each configuration. All blocks which have over 5000 total memory accessed are counted in the comparison. The values listed are the average absolute differences in hit rates of all counted blocks between the simulators driven by selectively dumped trace vs.
simulators driven by full memory trace. It will be seen that errors are very low for either method of Trace Approximation on these benchmarks against these caches.
Chapter 5

A Study of Cross-platform
Memory Trace-driven simulation

Memory tracing is not only limited by time and space constraints. Many times the feasibility is limited by the availability of the platform and tools. Trace-driven simulations are commonly used for evaluating proposed designs. The real hardware with proposed designs normally does not exist, therefore the memory traces have to be collected on currently available platforms. In addition, tracing memory, either to dump the trace or process the addresses on-the-fly, requires binary instrumentation tools. Such tools may not exist on the platform one wants to trace.

The lack of hardware and software leads to a very common practice in trace-driven simulation: cross-platform trace-driven simulation, which means traces are collected on an existing platform with available tools, but are used to simulate some different hardware designs. Such a cross-platform tracing scheme has serious problems. The memory traces collected on currently available hardware, compiled with a specific compiler, could be dramatically different from traces if they were collected on the hardware with proposed features, thus making the benefit claims of certain designs which are validated with cross-platform trace-driven simulation, doubtful.
In this chapter, this thesis studies the memory traces of several selected benchmarks on the following three platforms: Alpha EV67, IBM Power4 and Intel Itanium2. These benchmarks are compiled with highest optimization levels on each platform. They are then instrumented with different tools to trace the memory. Memory addresses are mapped back to source code at loop level. For each loop in the source code, the number of dynamic memory instructions are compared to see how many load/store instructions are required to performance the workload specified in that loop. In addition, this thesis studies the “similarity” of the addresses by simulating 25 different cache configurations. These are two of the major attributes used in the PMaC performance modeling framework.

The results are not surprising. The memory traces of the same applications on different platforms are “similar” but noticeably different at the same time. This chapter lists many architectural and compiler related reasons and proposed corresponding ways of factoring out the differences. This research also discovered that the total number of dynamic memory instructions corresponding to a given loop are likely to vary more than the simulated cache misses from the addresses corresponding to that loop. Although this chapter doesn’t lead to any concrete conclusion, it helps to raises alert about the widely used cross-platform trace-driven simulation approaches.

### 5.1 Snavely Conjecture

Dr. Allan Snavely\(^1\) came up with an interesting conjecture: the memory access patterns of applications are similar on different platforms from the performance prediction’s standpoint, if they are compiled with similar optimization flags. This thesis uses several experiments to verify the conjecture.

There are many reasons to believe that memory traces are inherently different on different machines. Differences in instruction sets, architecture prop-

\(^1\)Dr. Allan Snavely is the advisor of the author of this dissertation. He is also the creator and director of PMaC laboratory at the San Diego Supercomputer Center
erties, and compiler transformations all can affect how the binary accesses memory. For example, instruction scheduling done by the compiler can change the order of memory references, thus changing memory access patterns. However there are also many reasons one might believe memory traces are not that different. After all, given the same input, the binaries from the same source code are supposed to do the same work and produce the same output. While compilers differ, for a given code segment the choices for transformation are limited and it seems likely that many “wise” compilers behave similarly. Whether one believes that memory behavior on varying architectures is similar often depends on what kinds of questions one is interested in asking about an application’s memory behavior. For example, consider a simple program that sequentially scans a fixed-sized array. On one platform, the compiler reverses the access order while on another platform the original order is kept. If one is only interested in the size of memory region touched at run-time, the access patterns of the binaries on two platforms are essentially the same. However, if one is interested in exact stride information, the access patterns on two platforms are completely different.

5.2 Use Metasim Tracer to Detect Access Patterns

Metasim tracer was originally developed for Alpha processors using the ATOM [85] tool kit. It was later ported to Itanium processors using PIN [51], and Power processors using Dyninst [12]. Despite some differences in technical details of the instrumentation APIs, the underlying ideas and the analysis routines are the same across the APIs and platforms. Probes are automatically inserted into the binary to capture the effective addresses at runtime. The collected addresses are then processed on-the-fly to detect the memory access patterns. When the instrumented run completes, a concise report on access patterns is generated for each processor. Such reports are then used in the PMaC performance prediction framework [83].
The tracer has two modes, JustBB and Metasim. The JustBB mode only inserts one code snippet per basic block without acquiring and processing memory instruction effective addresses. This mode is considerably faster than full address tracing and enables one to calculate dynamic memory reference counts and to find important code sections quickly. The Metasim mode captures and analyzes memory effective addresses to detect access patterns in the stream of addresses.

Currently the analysis routines detect the following three characterizations of the effective addresses for each basic block

1. The dynamic memory reference count

2. Stride information (used to calculate the random ratio of the effective addresses)

3. The cache hit rates of the effective addresses on a variety of cache structures

The dynamic memory reference count reports how many loads and store instructions are executed (or issued) at runtime. These values can be gathered in JustBB mode, so they are the least expensive characterization. The other two characterizations require knowledge of actual effective addresses and require a Metasim mode run.

Cache performance is computed by simulating 25 different cache configurations, covering most current architectures and possible near future designs. Stride information and random ratio capture the spatial locality property of the address stream. Stride and random ratio information is gathered as follows. Each basic block has a local history window to keep track of the last 32 addresses. Every time an address is processed, it is compared to all the addresses in the history window to compute the minimum distance. If the distance is bigger than 128 bytes, it is regarded as random. In the final reports, there are counters for the random accesses and each stride accesses.

The PMaC performance modeling framework has been using these three memory access characters and patterns collected on Alpha processors to predict
the performance of other machines, even for systems that have not been built. This cross-platform prediction framework works relatively well – with an accuracy of about 80% [82]. An implicit assumption in this cross-platform prediction framework is that memory access patterns of the same application will be similar on different platforms, even on platforms supporting different instruction sets. But could some error be due to inaccurate memory modeling? One wonders how different the memory access patterns of the same application are on different platforms.

Acquiring dynamic memory addresses and detecting the access patterns are not trivial. Currently one sees 4 to 6 times slowdown on Alpha processors using ATOM. The slowdown on Itanium is more than 20 fold. In the case of DyninstAPI on Power processors, the overhead is so large that it is impractical to trace any real application. Table 5.2 shows the timings of our best implementations on each API. The slowdown is not only affected by the native overhead of each tools. It is also affected by the complexity of the ISA on the host platform on which one is tracing. DyninstAPI uses at least three JUMP instructions to install a probe. This native overhead is prohibitive for memory tracing. PIN is slightly more expensive than ATOM when naively instrumenting all memory instructions. However the simplicity of the Alpha instruction set enables certain static analysis techniques to reduce the number of memory instructions to be instrumented, thus reducing the overhead dramatically [28].

5.3 Experimental Setup

Metasim tracer reports the memory access patterns for each basic block. However, the basic blocks are not comparable on different platforms. This thesis maps the basic blocks to source code using the line numbers reported for the blocks. Such mapping is not always accurate, especially if the binary is compiled with high level optimization flags. To reduce the noise caused by inaccurate line numbers, a post-process script is used to parse the source code and merge the lines into
loops. The dynamic memory reference count of a loop is the sum of the memory references of all the basic blocks mapped to this loop. The cache performance and random ratio of a loop are a weighted average of corresponding data from the basic blocks mapped to this loop. Because source codes are the same on all platforms, the memory access patterns of the loops are used in this paper as the basis for comparisons.

The machines used in these experiments are listed in Table 5.1. Because the compiler is also a major factor in determining the memory access patterns, the compilation flags used on each machine are also listed. Basically one chooses the highest available optimization level hoping the compiler will reduce the number of unnecessary memory reference as much as possible. On Lemieux, the highest optimization level is -O5, but it generates incorrect results for SP, so -O4 is used. Several benchmarks from the NPB2.3 parallel benchmark suite [10] of different sizes and processor counts are selected in these experiments.

From Table 5.2, one can see overhead of capturing effective addresses is significantly higher than only instrumenting each basic block (Metasim versus JustBB). On Cheetah, neither LU.A.4 nor SP.A.4 completed Metasim runs in 5.5 hours and so slowdowns are calculated (cutoff and conservatively) using that value. The memory access patterns of these two applications on Power processors are interpolated based on what are reported before the jobs were killed.

<table>
<thead>
<tr>
<th>Machine</th>
<th>Processor</th>
<th>Tool</th>
<th>Compiler and Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>LemineX</td>
<td>Alpha EV67, 1GHz</td>
<td>ATOM V3.25</td>
<td>mpif77 -g3 -O4</td>
</tr>
<tr>
<td>(PSC)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cheetah</td>
<td>Power4, 1.33GHz</td>
<td>DyninstAPI 4.0</td>
<td>mpxlf -g -O5 -qstrict -qarch=pwr4 -qtune=pwr4</td>
</tr>
<tr>
<td>(ORNL)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Teragrid</td>
<td>Itanium 2, 1.3GHz</td>
<td>PIN 1.71</td>
<td>mpif90 -g -O3</td>
</tr>
<tr>
<td>(NCSA)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 5.2: Slowdowns of Different Tools
when instrumenting only basic blocks and when instrumenting all memory accesses. --- indicates the experiment could not complete in the given queue limit.

<table>
<thead>
<tr>
<th>Application</th>
<th>ATOM</th>
<th>PIN</th>
<th>Dyninst</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>JBB</td>
<td>Metasim</td>
<td>JBB</td>
</tr>
<tr>
<td>CG.A.4</td>
<td>1.89</td>
<td>38.23</td>
<td>8.55</td>
</tr>
<tr>
<td>FT.A.4</td>
<td>1.45</td>
<td>18.44</td>
<td>2.96</td>
</tr>
<tr>
<td>MG.A.4</td>
<td>1.40</td>
<td>30.00</td>
<td>3.91</td>
</tr>
<tr>
<td>LU.A.4</td>
<td>1.14</td>
<td>17.28</td>
<td>2.66</td>
</tr>
<tr>
<td>SP.A.4</td>
<td>1.30</td>
<td>18.67</td>
<td>3.28</td>
</tr>
<tr>
<td>CG.B.8</td>
<td>1.36</td>
<td>5.89</td>
<td>5.99</td>
</tr>
<tr>
<td>FT.B.8</td>
<td>1.31</td>
<td>3.59</td>
<td>2.62</td>
</tr>
<tr>
<td>MG.B.8</td>
<td>1.27</td>
<td>24.86</td>
<td>4.79</td>
</tr>
<tr>
<td>CG.C.8</td>
<td>1.35</td>
<td>4.06</td>
<td>6.14</td>
</tr>
<tr>
<td>FT.C.8</td>
<td>1.23</td>
<td>22.8</td>
<td>2.59</td>
</tr>
<tr>
<td>MG.C.8</td>
<td>1.18</td>
<td>7.56</td>
<td>3.17</td>
</tr>
</tbody>
</table>

5.3.1 Platform Specific Issues

In order to define a reasonable and rational comparison between binaries, one must address a number of architecture-specific issues.

RISC versus VLIW instructions

The ISA of the Itanium [38, 39] family is a VLIW (very long instruction word) and differs dramatically from the RISC style ISAs of Alpha [78], Power [88], and many other modern processors. One native Itanium instruction can specify tasks normally requiring several instructions on RISC processors. Each VLIW Itanium “instruction” is composed of up to 3, possibly independent, “operations” which are similar in power and organization to RISC-style instructions [38]. In the comparison, this thesis counts Itanium operations as “instructions” as these are more comparable in power to the Alpha and Power instructions. For example, the first instruction in Figure 5.1 will be counted as 2 loads and one floating point operation.
Figure 5.1: VLIW Instructions each made up of three operations

Filtering NO-OPs

One common issue with VLIW ISAs is that the compiler must use no-ops in the instruction when there are not appropriate operations to schedule. This effect is very obvious in the instruction counts. For example, no-ops account for 28% of the dynamic instruction (operation) count of cg.A.4. The percentage is similar for other applications. No-ops occur on other platforms as well. Particularly on Alpha, UNOP [78] has the form of a load instruction and ATOM regards such instructions as a load. In the results presented in this paper, all no-ops have been filtered and are not counted as instructions for any architecture.

Handling Itanium Specific Instructions

Itanium supports six types of “load pair” instructions. A load pair instruction can complete what would normally require two load instructions. For example, the ld fpd instruction loads two consecutive double precision data values, a total of 16 bytes, from the given address. These load pair instructions are treated as two normal load instructions in the simulator, mimicking the second load from the appropriate address.
One of the more unique features supported by Itanium and EPIC architectures is predicated execution. A predicated instruction can enter the execution pipeline before it is known whether, in fact, that instruction should really be executed. Each predicated instruction is tagged with a guarding predicate, a single bit value that must be evaluated to true or false before the point in the pipeline where that instruction changes program state. When the guarding predicate condition is false, the instruction is essentially transformed into a no-op, making no change to processor state. This can be used to schedule instructions in short branch sequences more efficiently in long pipelines, not requiring the branch to be resolved before instructions are fed into the pipeline. In the experiments, the guarding predicates are tested for each effective address. If the predicate is false, the address is discarded.

Itanium also support speculative loads. In the experiments, all speculative loads are discarded. The post analysis suggests such instructions are less common in the benchmarks selected in these experiments. Itanium also has several explicit prefetching instructions and the compiler uses such instructions extensively. In these experiments, all the explicit prefetched addresses are filtered.

5.4 Comparison of Memory Access Patterns

In this section, this thesis compares the three memory access patterns detected from the memory references of the same applications on different machines. This thesis performs three tiers of analysis at the loop level: performance on cache simulation, calculation of random ratios, and dynamic memory instruction count. From such comparisons, we hope to gain insights of how different the memory access patterns are on different platforms.
### Table 5.3: Cache Configurations Simulated and Reported

<table>
<thead>
<tr>
<th>Processor</th>
<th>L1</th>
<th>L2</th>
<th>L3</th>
</tr>
</thead>
<tbody>
<tr>
<td>altix1</td>
<td>size = 16KB, assoc 4way, line = 256KB, LRU</td>
<td>size = 256KB, assoc 8 way, line = 128B, LRU</td>
<td>size = 6MB, assoc = 12 way, line = 128B, LRU</td>
</tr>
<tr>
<td>Itanium</td>
<td>size = 256KB, assoc 8 way, line = 128B, LRU</td>
<td>size = 3MB, assoc 12 way, line = 128B, LRU</td>
<td>N/A</td>
</tr>
<tr>
<td>O3KA</td>
<td>size = 32KB, assoc 2 way, line = 32B, LRU</td>
<td>size = 8MB, assoc 2 way, line = 128B, LRU</td>
<td>N/A</td>
</tr>
<tr>
<td>O3KB</td>
<td>size = 32KB, assoc 2 way, line = 128B, LRU</td>
<td>size = 8MB, assoc 2 way, line = 128B, LRU</td>
<td>N/A</td>
</tr>
<tr>
<td>OPT1</td>
<td>size = 64KB, assoc 2 way, line = 64B, LRU</td>
<td>size = 1MB, assoc 16 way, line = 64B, LRU</td>
<td>N/A</td>
</tr>
<tr>
<td>Power 3</td>
<td>size = 64KB, assoc 128 way, line = 128B, random</td>
<td>size = 8MB, assoc 4 way, line = 128B, LRU</td>
<td>N/A</td>
</tr>
<tr>
<td>Power4-690b</td>
<td>size = 32KB, assoc 2 way, line = 128B, LRU</td>
<td>size = 720KB, assoc 4 way, line = 128B, LRU</td>
<td>size = 16MB, assoc 8 way, line = 512B, LRU</td>
</tr>
<tr>
<td>Alpha EV67</td>
<td>size = 64KB, assoc 2 way, line = 64B, LRU</td>
<td>size = 8MB, assoc 1 way, line = 64B, LRU</td>
<td>N/A</td>
</tr>
<tr>
<td>Xeon5</td>
<td>size = 16KB, assoc 4 way, line = 64B, LRU</td>
<td>size = 512KB, assoc 8 way, line = 64B, LRU</td>
<td>size = 1MB, assoc 8 way, line = 128B, LRU</td>
</tr>
<tr>
<td>Xeon6</td>
<td>size = 12KB, assoc 4 way, line = 64B, LRU</td>
<td>size = 1MB, assoc 8 way, line = 64B, LRU</td>
<td>N/A</td>
</tr>
</tbody>
</table>

#### 5.4.1 Cache Simulation Results Comparison

**Global Cache Simulation**

The overall cache hit rates (for the whole application) are very close, regardless of the trace platform and across all simulated caches. Figure 5.2 shows the overall cache hit rates of 5 NPB benchmarks of size A. For each application, the figure shows the simulated overall cache hit rates of 10 different cache con-
figurations of a range of modern processor designs. From left to right, they are altix1, it2a, o3ka, o3kb, opt1, pwr3, pwr4, sc45, xeon5 and xeon6 (described in Table 5.3). Although the architecture and compilers are different, and sometimes drastically different, the memory traces are “similar” in the sense that one can get very similar trace-driven cache simulation results from these traces taken on different machines.

Figure 5.2: Overall L1 Hit Rates from Alpha, Power4 and Itanium Traces on 10 simulated caches

Loop Level Cache Simulation Results Comparison

Figures 5.3, 5.5, 5.4 and 5.6 take a more detailed look at cache hit rates reported from Alpha traces and Itanium traces. The results show four sample
simulated caches based on the IBM Power3, IBM Power4, Alpha EV67 and Itanium 2 and include information from all levels of cache. Each data point is a loop from one of the NAS Parallel Benchmarks. The points are organized left to right by benchmark and within each benchmark from most frequently executed to least frequently executed. All loops that contribute at least 1% of the total dynamic instruction count of an application are shown. The difference in cache hit rates plotted on the y-axis is calculated by subtracting the cache hit rate of the Itanium trace from the cache hit rate simulated from the Alpha trace for the same loop.

Figures 5.7, 5.9, 5.8 and 5.10 show the same data for Alpha traces and Power4 traces. Generally, all the cache simulators report quite similar hit rates per
Figure 5.4: Difference of Loop Level Cache Hit Rates between Alpha and Itanium Traces: Simulating Cache Configuration of Itanium2
Figure 5.5: Difference of Loop Level Cache Hit Rates between Alpha and Itanium Traces: Simulating Cache Configuration of Power4
Figure 5.6: Difference of Loop Level Cache Hit Rates between Alpha and Itanium Traces: Simulating Cache Configuration of Alpha EV67
Figure 5.7: Difference of Loop Level Cache Hit Rates
between Alpha and Power4 Traces: Simulating Cache Configuration of Power3
Itanium 2

Figure 5.8: Difference of Loop Level Cache Hit Rates between Alpha and Power4 Traces: Simulating Cache Configuration of Itanium2
Figure 5.9: Difference of Loop Level Cache Hit Rates between Alpha and Power4 Traces: Simulating Cache Configuration of Power4
Figure 5.10: Difference of Loop Level Cache Hit Rates between Alpha and Power4 Traces: Simulating Cache Configuration of Alpha EV67
loop from the memory traces across the different binaries (Alpha, Itanium, IBM Power 4). On a large majority of loops, the difference in cache hit rate generated by each binary stream generally differs by less than +/- 5%. However the relative difference in cache miss rates can be significant. Different users may interpret the results differently: the differences could be regarded as either marginal or dramatic depending on how the cache simulation results are used in actual analysis.

5.4.2 Comparison of Loop Level Random Ratios

In the Metasim convolver, which has been used to make performance predictions of scientific parallel applications, a random ratio is defined and that value is used to select between one of two measured curves of cache hit latency. In general, it is discovered that significant accuracy is achieved from categorizing each basic block as either “random” or “strided” in its access pattern. A value of 0.2 has been experimentally identified as a good cut-off of randomness to accurately model memory latency. That is, if a basic block has more than 20% of its memory accesses made in a non-strided way, one uses measured random memory latency and bandwidth values.

In Figures 5.11 and 5.12 each loop is represented by a point plotted at the x and y coordinates determined by its random ratio from an Itanium (or Power 4 for Figure 5.12) trace and from an Alpha trace, respectively. These figures show that few loops have a random ratio value that would put it in a different “randomness class” as determined by traces on different architectures. That is, while the exact random ratio may differ slightly on many loops depending on the architecture stream utilized, very few cases occur where a loop from an Alpha binary has a random ratio above 0.2 and that same loop from an Itanium binary has a random ratio below 0.2 (and vice versa). Only those few points that lie in the regions marked A and B have random ratios that would reflect different behavior across the two machines traced. Once again, traces from different platforms show certain similarities even if they are collected on different traces. For the performance
prediction based on Metasim tracer, the differences are not a big issue because of the selected threshold. However, the differences in random ratio could be disastrous for other analysis.

5.4.3 Dynamic Memory Reference Comparison

Interestingly, there is much greater variability in dynamic load and store counts per loop across architectures than there is in performance of those loops as measured by cache simulation or random ratio. Additionally, variation is more pronounced for certain benchmarks. Nonetheless, 45% of loops reported on have nearly the same dynamic load count (a ratio between 0.9-1.1) and 73% of loops have nearly the same dynamic store count.

Figure 5.13 shows the difference per loop in the dynamic count of loads and stores executed per loop for an Itanium 2 platform in relation to the baseline Alpha platform. Figure 5.14 shows similar results comparing a Power 4 platform to an Alpha. In all graphs, the x-axis shows loops organized by benchmark, within each benchmark, loops are ordered based on their frequency (from left to right). Only those loops which contribute more than 1% of all dynamic instructions in the benchmark are included to improve readability.

Due to the noticeable difference in dynamic memory reference count, traces collected from significantly different platforms could cause error in memory performance models which use dynamic memory reference count as a major component of the model.

5.5 A Consideration of the Differences in Memory Access Patterns

There are many possible reasons why the memory access patterns of the same applications are different on different machines. Some of these differences are attributable directly to significant differences in the ISAs studied. This is
Figure 5.11: Random Ratio: Alpha vs. Itanium

Figure 5.12: Random Ratio: Alpha vs. Power4
Comparison of Dynamic Memory Reference Count

Figure 5.13: Difference in Dynamic Memory Reference Counts: Alpha vs. Itanium

particularly the case for Itanium. Additionally, other differences emerge that are
compiler dependent, though many of those compiler issues are also related, in
varying degrees, to architecture-level issues. Due to space limitations, this thesis
only very briefly outlines some of the major issues here.

1. Special instructions

2. Different number of registers

3. Register spill and fill

4. Function calls inside loops

5. Cross loop transformation and optimization

6. Compiler inserted prefetching loads
7. Redundant loads and stores

The special instructions on each platforms are easy to detect and work around. This thesis explained how load pair instruction, prefetching instructions, speculative load instructions, and predicated instructions are handled in Section 5.3.1. These workaround approaches are aimed to provide a fair comparison basis.

The number of available registers has a huge impact on how compilers generate the binary codes for a given source code. Insufficient registers can cause excessive load and store references on the stack. The number of registers also affects how the parameters can be passed, through the registers or the stack. Itanium has many more registers than Alpha and Power 4. From Figure 5.13, one can clearly
see the reduced number of loads needed on Itanium for many loops which comes from Itanium’s ability to hold a large number of values in registers. Loop level transformation and optimization done by the compilers also affects the memory access patterns of the loops. Compilers can fuse several loops [44, 3, 57] to achieve better instruction level scheduling and better register reuse, among other things. When loops are fused, it is possible that some load and store instructions can be eliminated. This is most notable in cases where the first loop calculates some intermediate results which are used by the second loop. That is the reason why some loops have no load or store references on Itanium (note the number of data points with an X axis value of 0 in Figure 5.13). Compilers can also insert prefetching loads and store hints to make the address stream more friendly to the cache structure of the host machine. Unlike explicit prefetching instructions, such prefetching loads are just like normal load instructions and more difficult to detect and remove from the address stream.

Even though the highest optimization levels have been used on all three platforms, there are still redundant loads and stores in certain loops. Each compiler appears to have strengths and weaknesses in different scenarios.

Although all the reasons listed above can also affect the simulated cache results and random ratios of the loops, the real impact is much less noticeable. For each reason, one could have a workaround solution to make the memory access counts more similar, but it would be difficult to guarantee that all possible scenarios would be caught and addressed.
Chapter 6

Related Work

6.1 The PMaC Performance Modeling and Prediction Framework

The work presented in this thesis is part of a collaborative effort to build a performance modeling and prediction framework for large scientific applications. Figure 6.1 lists the major components of this framework and their interactions. This framework uses two models, i.e., single-processor model and communication model, to predict the execution time of parallel applications. In each model, several parameters called Hardware Profiles are used to summarize the most important attributes of the system related to its performance. In the single processor model, a hardware profile consists of effective bandwidths of every level of the memory hierarchy, the number of functional units and the peak FLOPS etc. The hardware profile for the communication model includes effective bandwidths of various network components. These profiles can be either measured using the PMaC micro-benchmark suite [87] on currently available machines or determined from published or proposed specifications. Application signatures are used to collect and summarize the important attributes and patterns of an application’s dynamic executions. In the communication model, application signatures focus on the communication activities (such as MPI calls) of the applications. These signatures
are collected and characterized using Dimemas [9, 33]. Application signatures in
the single processor model consist of various dynamic counters, dynamic access
patterns and some static attributes. These attributes are measured using Metasim
tracer and explained in detail in the next section.

After machine profiles of the target system and application signatures of
application have been acquired, they are fed into the convolver for performance
prediction. The details of the convolution methods are beyond the scope of this
thesis and can be found in [17, 16].

6.2 Metasim Tracer

Metasim tracer is the tool in the PMaC framework to acquire application
signatures for the single processor performance model. It first identifies the follow-
ing static attributes via static analysis of the binaries. Such attributes require no
dynamic tracing and are very inexpensive to acquire.

1. **FP/MEM ratio**

   This ratio indicates on average how many load/store operations are required
   for each FP operating in the instruction mix of the given basic block. The
   ratios are used by the convolution method to determine whether FP units or
   memory units are the bottleneck if the given block executes on a particular
   machine.

2. **Branch Intensity**

   This ratio indicates how many valid operations, such as FP and memory
   operations, the block has between each branch instruction. These numbers
   are used by the convolution methods to determine the steady states of the
   functional units.

3. **Dependency Distance**

   This number indicates the instruction level parallelism (ILP) of the given
Figure 6.1: PMaC Performance Prediction Framework

This thesis work is mostly related to collecting Application Signature on single processors (shaded)

instruction sequence. If this number is 1, the given sequence is inherently sequential so extra functional units in the hardware will idle. The convolution
methods use these numbers to fine tune the predictions.

Metasim tracer collects several dynamic counters, such as total floating point operations, total memory operations etc. Metasim tracer acquires these counters using lightweight instrumentation at the basic block level. Once we know how many times each block is visited, the dynamic counts of various operations can be calculated by multiplying the block visit counts by the static operation counts in these blocks.

Finally, Metasim tracer spends most of its time detecting each application’s memory access patterns. There are two sets of data in the memory access patterns.

1. Random Ratio

Random Ratio is used to show how random or sequential the dynamic addresses are for each basic block. This ratio is used by the convolution method to pick the expected effective memory bandwidth. The routine which calculates random ratios keeps a global observation window of captured addresses. Each newly captured address is compared to the addresses in the window to determine the minimum distance. If this distance is more than a given threshold (for example, 128 bytes), this address will be classified as random, and the counter of the basic block where this address comes from will be incremented. The observation window is updated in FIFO (first in, first out) order.

2. Cache Hit Rates

The current convolution method largely uses a linear memory model described in Chapter 1 to predict memory performances. Thus getting cache hit rates of the targeting machine is one of the major tasks in performance prediction. A parallel cache simulator is implemented to simulate 25 cache configurations shown in Table 6.1 and Table 6.2 all at once. These configurations cover most of available processors and several possible future designs.
Hardware prefetching is also simulated according to the cache configuration.

These ratios and cache hit rates are acquired with low level instrumentation. Metasim tracer instruments several analysis functions and simulators into the applications’ binaries using binary instrumentation tools. These analysis routines and simulators capture the effective addresses at runtime and process them without saving any addresses. When the instrumented binary completes, a unique report file will be generated for each processor. These report files are then used by the convolution method to predict single processor performance. Collecting dynamic memory access patterns is by far the most expensive component; how to reduce the cost is the focus of this thesis.

Table 6.1: Cache Configurations Simulated: part 1

<table>
<thead>
<tr>
<th>Processor</th>
<th>L1</th>
<th>L2</th>
<th>L3</th>
</tr>
</thead>
<tbody>
<tr>
<td>altix1</td>
<td>16KB, 4way, 256B, LRU</td>
<td>256KB, 8way, 128B, LRU</td>
<td>6MB, 12way, 128B, LRU</td>
</tr>
<tr>
<td>Itanium</td>
<td>256KB, 8way, 128B, LRU</td>
<td>3MB, 12way, 128B, LRU</td>
<td>N/A</td>
</tr>
<tr>
<td>IT2B</td>
<td>256KB, 8way, 128B, LRU</td>
<td>4MB, 16way, 128B, LRU</td>
<td>N/A</td>
</tr>
<tr>
<td>O3KA</td>
<td>32KB, 2way, 32B, LRU</td>
<td>8MB, 2way, 128B, LRU</td>
<td>N/A</td>
</tr>
<tr>
<td>O3KB,</td>
<td>32KB, 2way, 128B, LRU</td>
<td>8MB, 2way, 128B, LRU</td>
<td>N/A</td>
</tr>
<tr>
<td>OPT1</td>
<td>64KB, 2way, 64B, LRU</td>
<td>1MB, 16way, 64B, LRU</td>
<td>N/A</td>
</tr>
<tr>
<td>Power3</td>
<td>64KB, 128 way, 128B, LRU</td>
<td>8MB, 4way, 128B, LRU</td>
<td>N/A</td>
</tr>
<tr>
<td>Power4-690b</td>
<td>32KB, 2way, 128B, LRU</td>
<td>720KB, 4way, 128B, LRU</td>
<td>16MB, 8way, 512B, LRU</td>
</tr>
<tr>
<td>Power5</td>
<td>32KB, 4way, 128B, LRU</td>
<td>960KB, 1way, 128B, LRU</td>
<td>18MB, 12way, 512B, LRU</td>
</tr>
<tr>
<td>Alpha EV67</td>
<td>64KB, 2way, 64B, LRU</td>
<td>8MB, 1way, 64B, LRU</td>
<td>N/A</td>
</tr>
<tr>
<td>Xeon5</td>
<td>16KB, 4way, 64B, LRU</td>
<td>512KB, 8way, 64B, LRU</td>
<td>1MB, 8way, 128B, LRU</td>
</tr>
<tr>
<td>Processor</td>
<td>L1</td>
<td>L2</td>
<td>L3</td>
</tr>
<tr>
<td>------------</td>
<td>-------------------</td>
<td>-------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>system 01</td>
<td>32KB, 2way, 128B,</td>
<td>768KB, 4way, 128B,</td>
<td>16MB, 8way, 512B,</td>
</tr>
<tr>
<td></td>
<td>LRU</td>
<td>LRU</td>
<td>LRU</td>
</tr>
<tr>
<td>system 02</td>
<td>32KB, 2way, 128B,</td>
<td>768KB, 8way, 128B,</td>
<td>16MB, 8way, 512B,</td>
</tr>
<tr>
<td></td>
<td>LRU</td>
<td>LRU</td>
<td>LRU</td>
</tr>
<tr>
<td>system 03</td>
<td>64KB, 2way, 64B,</td>
<td>1MB, 16way, 64B,</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>LRU</td>
<td>LRU</td>
<td></td>
</tr>
<tr>
<td>system 04</td>
<td>256KB, 8way, 128B,</td>
<td>6MB, 12way, 128,</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>random</td>
<td>LRU</td>
<td></td>
</tr>
<tr>
<td>system 05</td>
<td>8KB, 4way, 64B,</td>
<td>512KB, 8way, 128B,</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>LRU</td>
<td>LRU</td>
<td></td>
</tr>
<tr>
<td>system 06</td>
<td>64KB, 2way, 64B,</td>
<td>8MB, 1way, 64B,</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>LRU</td>
<td>LRU</td>
<td></td>
</tr>
<tr>
<td>system 07</td>
<td>128KB, 2way, 64B,</td>
<td>1MB, 16way, 64B,</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>PLRU</td>
<td>LRU</td>
<td></td>
</tr>
<tr>
<td>system 08</td>
<td>128KB, 8way, 64B,</td>
<td>1MB, 8way, 64B,</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>LRU</td>
<td>LRU</td>
<td></td>
</tr>
<tr>
<td>system 09</td>
<td>256KB, 8way, 128B,</td>
<td>12MB, 12way, 128B,</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>PLRU</td>
<td></td>
<td></td>
</tr>
<tr>
<td>system 10</td>
<td>64KB, 4way, 128B,</td>
<td>1920KB, 10way, 128B,</td>
<td>18MB, 12way, 256,</td>
</tr>
<tr>
<td></td>
<td>PLRU</td>
<td>PLRU</td>
<td>PLRU</td>
</tr>
<tr>
<td>system 11</td>
<td>256KB, 8way, 128B,</td>
<td>6144KB, 6way, 128B,</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>PLUR</td>
<td>PLRU</td>
<td></td>
</tr>
<tr>
<td>system 12</td>
<td>16KB, 8way, 128B,</td>
<td>2MB, 8way, 64B,</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>LRU</td>
<td>LRU</td>
<td></td>
</tr>
<tr>
<td>system 13</td>
<td>16KB, 2way, 32B,</td>
<td>512KB, 4way, 32B,</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>LRU</td>
<td>LRU</td>
<td></td>
</tr>
<tr>
<td>system 14</td>
<td>64KB, 4way, 64B,</td>
<td>2MB, 8way, 128B,</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>LRU</td>
<td>LRU</td>
<td></td>
</tr>
</tbody>
</table>

### 6.3 Binary Instrumentation Models

Binary instrumentation tools are a theme of this thesis. An assembly rewriter is introduced in Chapter 3 to demonstrate an asynchronous instrumentation scheme for lightweight instrumentation. The other work presented in this thesis either uses binary instrumentation tools, or deals with reducing the overhead of using these instrumentation tools for memory tracing. Many of the strategies shown in this thesis are largely determined by the nature of binary instrumentation.
tools used. So in this chapter, the basic concepts and implementations of binary instrumentation tools are briefly introduced.

Instrumentation tools are used to insert probes or analysis functions into applications so dynamic the behaviors can be observed, traced and analyzed. Instrumentation can be done at different levels including source codes, assembly codes and binary codes. Binary instrumentation tools work on binary files on disk or binary images in memory, and by far are the most commonly used tools for instrumentation because of the three major advantages.

1. Binary instrumentation causes less distortion to applications dynamic behaviors. The instrumentation is performed after compilers have completed all the transformations and optimizations. Inserted instructions can be transparent to original instructions, so the observed access patterns or behaviors are more likely to be the original ones. On the other hand, instrumented statements at the source file level are up to compilers’ transformations and optimizations. They could be moved to different locations or completely removed. Programmers normally rely on locks to make sure instrumented statements stay where are intended to be. However these locks could have side effects on the compilers in issuing instructions for the original statements. These side effects could make the observed behaviors different from what it should have been without the instrumentation.

2. Binary files are generally more accessible than source files. Many commercial packages and applications are distributed in binary. It is impossible for a general user to gain access to the source codes files of these applications. This leads to instrumenting binary to be the only viable approach for acquiring dynamic information from them.

3. Binary instrumentation is more flexible than other instrumentation schemes. Instrumentation at source code level or assembly level is inherently static. All the instrumentation has to be done before executing the binary, and the
inserted codes will remain in the new binary all the time.

Instrumentation tools working at the assembly level are mostly prototypes. They are less intrusive because most optimizations and code transformations have been performed by compilers while generating the assembly codes. These tools are also easier to implement because there are much less rules needed to create a correct assembly file than a working binary. For example, targets of branch instructions can be accessed using labels in assembly files, and these labels will be automatically calculated and filled in by assemblers and linkers. Tool writers will have to carefully implement such functionality to correctly manage all the offsets in the new binary if they choose to create an instrumentation tool that works on binary directly.

Depending on when the binary is examined and instrumented, binary instrumentation tools have three implementation models: static, just-in-time (JIT) and dynamic. Binary instrumentation tools working on static binary images on disk are also referred as binary rewriters. These tools create new, instrumented binaries independent to the original ones. Figure 6.2 shows an example of how binary rewriters create a new binary. Binary rewriters have the most flexibility in creating instrumented binaries because there is practically no limitation on how a new binary should be generated as long as it keeps the correctness of the original. Although most tools will only insert a function call at each instrumentation point, it is possible for a binary rewriter to inline all instrumented functions and perform advanced cross-routine optimizations. Tools implemented based on this model include EEL [48], Etch [69], MORPH [99] and ATOM [85].

Binary rewriters generally introduce minimal overhead into the instrumented binaries. However, there are two major drawbacks of binary rewriters. Firstly, more and more application now mix data and code in their binaries, and a static tool may not be able to distinguish the two. Secondly, once the new codes are inserted, it will very difficult to change or remove them from the new binaries.

Dynamic instrumentation tools examine memory images after binaries have been loaded into the memory. It is unable to expand the memory image like
Figure 6.2: Static Binary Rewriter Model

binary rewriters could with static images on disk since many system tables, such as page table etc, have been set and they are very difficult to change. These instrumentation tools are also referred as patching tools, emphasizing the fact that they have to modify binaries in place. Figure 6.3 illustrates how binary patching tools work. Some of the original instructions are replaced with a jump instruction. This jump instruction is set to point to a newly allocated space where the replaced instruction sequence and the instrumented code will reside. Another jump instruction will be added at the end of the instrumented code sequence so the execution can return to where it originally comes from. Binary patching in this manner is very flexible in modifying the instrumentation since jump instructions can be set to point to other code sequences, for example a different function entry, even while the program is running. These jump instructions can also be replaced with the original instruction sequences so, effectively, the instrumented snippets can be dynamically removed. Tools are implemented based this model include Dyninst [12], Vulcan [84] and DTrace [15]. The overhead introduced by dynamic instrumentation tools is significantly higher. Having at least two jumps per each
instrumentation point\textsuperscript{1} makes such tools less practical for fine-grained instrumentation tasks such as instrumenting memory instructions. Furthermore, it is not always possible to find enough instructions to replace, particularly on platforms with variable instruction length such as Intel IA32 [40] since multiple instructions might need to be replaced for one jump instruction.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{dynamic_patching.png}
\caption{Dynamic Binary Patching Model}
\end{figure}

Just-in-time can be regarded as an advanced implementation of dynamic instrumentation. The system maintains a small buffer called code cache which holds the actual instruction sequences to be executed (inserted and original). It is called Just-in-time because sequences of instructions are examined and instrumented the first time they are encountered. The instrumented sequences are then kept in the cache for future execution. Figure 6.4 shows the implementation of a JIT instrumentation tool. Compared to binary patching tools, JIT instrumentation tools are much less restrained because instruction sequences are kept in code cache and controlled by the instrumentation tool itself. JIT instrumentation tools can also remove the instrumentation by flushing the instrumented instruction sequence in the code cache and replacing them with original ones. Tools are im-

\textsuperscript{1}Two jumps are the minimum requirement. In most implementations, three or more jump instructions are needed.
plemented based on this model include Valgrid [61], Strata [74], DynamoRIO [11] and PIN [51, 37].

JIT tools are not just instrumentation tools. Generally, they have to include complete run-time support systems so the execution can be forced to fetch instructions from the code cache. This way the contents in the code cache can be modified without affecting other systems.

![JIT Instrumentation Model](image)

Figure 6.4: JIT Instrumentation Model

### 6.4 Other Related Work

Throughout this thesis, related work has been discussed in each related section. This chapter presents a structured overview of the related work. These works are summarized into the following categories:

#### 6.4.1 Trace-driven Simulation

Trace-driven simulation [93, 102] has been an important tool in helping design memory systems and system management strategies. Much work on memory system design [86, 80, 101] relies heavily on memory trace-driven simulation.
Uhlig and Mudge [90] presented a thorough survey covering many aspects of trace-driven memory simulation. In most work [91] [80] [101], the systems where the traces are collected are nearly identical to the systems which the simulations are targeting for, excepting differences in memory sub-system. In these cases, it is relatively safe to assume the collected traces will be representative of the target system. Traces that drive simulations are critical to the correctness of the results and conclusions. Yet, incomplete, even incompatible traces can be better than nothing [32, 27]. Kaplan [42] argues that whole system traces should be collected for simulating virtual memory management. Goldschmidt and Hennessy [34] discuss the possible inaccuracy related to timing issues in parallel environments, and suggest those performance metrics related to synchronization latency should be predicted based on timing-independent traces. In the PMaC performance prediction framework [82, 83, 17, 16], trace-driven simulation is used to summarize application signatures, which are later used by convolution methods to predict the run time of applications.

6.4.2 Capturing Addresses

Trace-driven simulation requires tools and methods to captures the traces. Agarwal et.al [1] introduced an efficient method of capturing address traces using microcode supported by the VAX1 8200 processor. Most recent research relies on instrumentation tools to collect traces. Much work on cache-conscious data layout [14, 70], software prefetching [21], and data locality optimization, relies on the ability to instrument binaries to acquire effective address traces. However the slowdown caused by binary instrumentation often means such work is restricted to kernels and very short programs. Work related to instrumentation tools is presented in Chapter 6.3.
6.4.3 Sampling

Sampling [95, 25, 64] has been shown to be extremely effective in processing event traces. Due to the repetitive nature of many event traces, processing only a small portion of the elements in traces can yield the same or very similar results compared to processing full traces. Periodic sampling is the easiest to implement. A global variable is used to control whether addresses should be processed or discarded. This variable is turned on and off periodically based on time intervals or event counters. In most cases, sampling is turned on and off rather arbitrarily. The decision as to whether addresses should be processed are unrelated to the applications’ behaviors and this fact can cause large unpredictable errors. The only way to reduce error is to increase sampling frequency, but this will definitely increase the overhead as well.

Many phase-detection mechanism [66, 49, 58, 76] and phase-guided sampling techniques [67, 50] have been proposed. Among them, SimPoint [77] determines sampling by observing application behaviors and phases. This generally leads to more representative sampling. SimPoint-guided sampling is adopted in this work [50].

When sampling is used in memory tracing, generally there is still significant overhead for capturing addresses even if the addresses are determined to be discarded without processing (sampling is off). Arnold and Ryder [6] presented a novel implementation to remove these costs. The overhead when sampling is off can be removed by duplicating the binary, one instrumented and one uninstrumented. Sampling is turned on and off by switching between two binaries. This technique is extremely effective and has been applied extensively [24, 21].

6.4.4 Trace Representation

How the trace are represented and stored not only affects disk space requirements, it also determines how the traces can be used. Larus presented Whole Program Paths (WPP) as a compressed representation of the entire applications’
execution traces, including call graphs, branch targets etc. WPPs can be analyzed offline to detect patterns such as repetition of blocks, phase changes etc. Many profile based optimizations used WPP to represent traces [20, 56, 70]. Enriched WPPs have also been proposed to include other information such as dependency information in additional to control flow information originally focused on by WPP. Zhang and Gupta [100, 97, 98] presented time-stamped WPP (TWPP) aimed at providing easy access to path traces by restricting the paths on function level basis. All available compression schemes focus on detecting repetition and patterns from a dynamic sequence of events.

6.4.5 Lossless Trace Compression

Traces are compressed to reduce storage requirements. Compression generally requires traces can be restored exactly from their compressed presentations. If the process to reduce trace size is not reversible, it is generally refer to as trace reduction. Data compression is a major field in computer science. Here we only focus on compression schemes that have been used for compressing event traces. Gzip [35] and the utility functions provided in zlib [104] are implementations of LZW [103] algorithm. The LZW algorithm is based on Huffman coding and it is a very generic compression algorithm, useful for text files and binary files. Gzip is widely used to compress trace files because it is extremely fast and easy to use. It also supports incremental compression so no scratch disk is required.

The Sequitur [63, 62] algorithm was designed to detect repetitive patterns in sequences. It is also a generic algorithm which has been widely used in compressing such things as music scores and DNA sequences. It was also used with Whole Program Paths to compress control flow traces. Sequitur compresses sequences by building a context free grammar from the sequence. The detailed implementation is further discussed in Chapter 4.1.2 to motivate Path Grammar Guided Trace Compression (PGGTC) compression scheme. Sequitur can achieve very high compression ratio when used to compress control flow traces. However
it requires scratch disk to save the trace first and it can be extremely slow.

VPC is a value predication-based compression scheme proposed by Burtscher and Jeeradit [22, 13]. It is designed specifically for compressing execution traces and requires certain formats to work. The published compression ratios are quite good. VPC uses several value predictors to predict next value based on the observed traces. This predicated value is then compared to the real captured value. If they are the same, one mark is stored in the trace to indicate the correctness of prediction. Otherwise the actual value is stored in the compressed trace. The exact same predictor is used in decompression.

Memory traces are more difficult to compress because there are often much less obvious repetitions in the elements. Nevertheless various research has proposed to compress the address sequences losslessly [41, 52, 55, 22, 24]. Sequitur and VPC have also been used for effective address compression. Among recent research, Luo and John proposed Locality-based online trace compression (LBTC) [52]. LBTC uses a small cache to detect temporal locality in the trace. If the memory reference hits in the cache, the "static" attributes related to the addresses are retrieved from the compression cache without being stored in the trace file. The compression cache is indexed with the physical address of the instruction fetches.

Stream Based Compression (SBC) [55] exploits streams in instruction sequences and uses streams to compress instruction traces and data address traces. An instruction stream is a sequential run of instructions, from the target of a taken branch to the first taken branch in sequence. A stream table keeps relevant information about streams: starting address, stream length, instruction words and their types. All instructions from a stream are replaced by its index in the stream table. The data address stride and the number of stride repetitions is attached to the corresponding instruction streams and stored separately.
6.4.6 Lossy Trace Size Reduction

Saving full memory reference traces on disk is not a realistic option even for small benchmarks, especially if the traces have substantial random elements.

Many techniques have been proposed to reduce the amount of data stored by discarding most data while still keeping remaining data representative of the full traces [91, 96]. Sampling techniques can be applied with minor changes to reduce trace sizes. Only the addresses captured when sampling is on are saved, the others are discarded. The total reduction in trace size depends how many addresses are captured during the sampling periods. Toomula [89] proposes using sampling to collect memory skeleton to represent full traces.

More advanced implementations discard non-critical addresses to save space [2, 23] through filtering. Filters are simple simulators that are used to determine whether the given address is “expected” or not. If it is expected based on previously observed events, the address are discarded.

Historically, filtering techniques focus on preparing traces for page level simulations, such as for simulating page table replacement policies. Generally filtering techniques discard too much information to be accurate for cache simulations [23]. Filtering techniques are not effective for reducing random traces because addresses all look “important” and unpredictable.
Chapter 7

Future Work

In this chapter, some open questions and possibilities related to memory tracing are discussed.

7.1 Broader Applications of the Techniques for Reducing Time Overhead

The work presented in this thesis originated from trace-driven simulators used in the PMaC performance modeling and prediction framework. How to extend the solutions presented in this thesis beyond the realm of trace-driven simulation is an interesting question. Many debugging tools are too painfully slow to be usable if memory accesses are monitored. Most profiling schemes are too slow to be feasible for online optimizations. Therefore the buffering techniques, static analysis and delayed instrumentation techniques presented in Chapter 2 could be useful for reducing the time overhead of these memory tracing tasks.

7.2 Easier to Use Asynchronous Binary Instrumentation Tools

This thesis introduces an asynchronous instrumentation scheme specially designed for fine-grained tracing tasks such as memory tracing. The current imple-
mentation, ALITER, is created as an assembly rewriter. It was easier to implement but it is not exactly user-friendly. A true binary rewriter based on similar asynchronous instrumentation schemes would be very useful.

7.3 Broader Applications of Asynchronous Schemes

An asynchronous tracing scheme is presented in this thesis to match the asynchronous nature of many driven-driven applications. The benefits are substantial. Can similar asynchronous schemes be created for traditionally time-critical tasks such as debugging? With the ability of execution roll-back, asynchronous tracing for debugging might be made feasible. Such non-traditional combination could turn out to be very efficient in reducing the time cost.

7.4 Faster Path Grammar Guided Trace Compression

Although PGGTC is much faster compared to Sequitur, it is still relatively slow. Such slowness limits the applications one can use PGGTC for compressing on-the-fly in the given time limit. Much of the slowness is caused by the actions to maintain the execution stack so the online analysis could identify which structure the current path belongs to. Is it possible to make the stack maintenance less expensive? Is stack even necessary to compress a trace? Further profiling and analysis might be helpful to answer these questions.

7.5 Better Trace Statistics for Approximation

Trace approximation proposed in this thesis has been shown extremely effective in reducing sizes of memory traces which contain many random elements. Are there better statistics which are easier to collect as well? Is it possible to estimate the quality of collected statistics if either time or space is limited?
7.6 More Usages of Structure Path Histories

In this thesis, structure path histories are presented as by-products of PG-GTC. Many unique features of such a representation makes it a good candidate for representing future execution traces. This work has explored several advanced usages of such representations including trace comparisons and artificial trace alternations. Much more work would be needed to fully appreciate the potential of this trace representation.

7.7 Further Studies on Cross-platform Tracing

Despite the fact it is error-prone, cross-platform trace-driven simulation is still a popular approach. Sometimes, it could be the only solution. Although it is unlikely to find a complete set of ways of factoring out all the differences of different platforms, the ability of quantitatively evaluating the difference will be good enough for many cases. If one can ascertain that the error caused by differences of platforms is less than 3%, then 10% improvement indicated by cross-platform trace-driven simulation would still be convincing.

7.8 More Resources for Memory Tracing

Instrumentation is currently the mostly frequently used method for collecting memory traces. Although many solutions, either from tool users’ standpoint or from tool writers’ standpoint, have been introduced in this thesis, memory tracing is still very costly in time. Shared cache and multiple-core processor design could provide one short-cut for memory tracing. One dedicated core could be used to monitor the memory addresses issued from other cores, thus address collection could be implemented without extra cost.
Chapter 8

Conclusion

This thesis has presented solutions and techniques to reduce the tremendous time overheads and space requirements related to memory tracing, so that memory access behaviors of larger, parallel applications can be efficiently traced and studied.

In practice, collecting and processing of memory traces of large, parallel applications, it is not always about being perfect in every aspect, or even in any aspect. Instead, a successful solution is finding a balance in the time and space on available platforms with available tools. The requirement for a balanced solution is indeed another challenge in memory tracing. Many solutions in this thesis, especially when dealing with space, accuracy and availability, are trade-offs among many factors. The idea behind these solutions is to get an intermediate representation of the memory trace, which is small enough to be stored on the disk, accurate enough for trace driven simulations and fast enough to collect and process.

Specifically this thesis has made the following contributions:

1. The time overhead introduced by binary instrumentation is analyzed quantitatively. Based on such analysis, techniques including asynchronous processing via buffering, static analysis, symbolic execution and delayed instrumentation are explored and introduced to reduce the time cost of memory tracing. The specific result reported on is a vastly faster Metasim tracer with
which it is now feasible for ordinary users of HPC system to memory-trace their applications. Additionally, high-end users such as HPCMO will save approximately 80% of their previous tracing costs.

2. This thesis introduced an asynchronous tracing scheme. A lightweight instrumentation tool, ALITER, was also presented to demonstrate the advantages of asynchronous instrumentation schemes in reducing tracing time. Overall, ALITER only causes less than a 2-fold slowdown to collect full memory traces of the selected benchmarks; this contrasts with ten and even several hundred fold slowdown if generally available instrumentation tools are used.

3. Path Grammar Guided Trace Compress (PGGTC) is designed to speedup compression process and reduce trace sizes. The efficacy of low-level general purpose compression schemes, such as gzip, is greatly enhanced when traces are transformed by PGGTC based on program structure and phases. Combined with gzip, PGGTC can compress control flow traces over 330 times smaller. Compared to the widely popular Sequitur algorithm, PGGTC with gzip is on average 40 times faster, while the traces are only 3 times bigger. When used with Sequitur, PGGTC can double the compression ratios of Sequitur and do it 14 times faster.

4. This thesis introduced Structured Path Histories (SHP), a new representation format for control flow traces. SPH overcomes several major problems of Whole Program Paths(WPP). Besides being easy to use, SHP enables many advanced features, such as direct comparisons between traces of the same application from different runs, artificially altering part of the traces, and generating full traces without actual tracing. These features could be valuable for studies in architecture design and performance optimizations.

5. Trace Approximation (TA) is introduced to reduce the size of effective addresses traces with significant randomness. Such traces are often impossibly large even after being compressed with any lossless scheme. TA is designed
to capture the locality properties and other important statistics of the traces, instead of the exact random events themselves. The recorded statistics, which have been shown effective in generating traces very similar to original ones, only require orders of magnitude less disk space to store.

6. Cross-platform simulation and related issues have been studied in this thesis. The study discovered many reasons that can cause the same application to have different memory access patterns on different platforms. Even after many ways of factoring out the differences have been suggested and implemented, memory traces still show significant differences. How to interpret such variations depends on the nature of different users’ analysis; this thesis raises the issue and hope future simulators using cross-platform memory traces are used with care.

With all the techniques, schemes and workarounds, this thesis has made it feasible to trace memory accesses of large parallel applications. These solutions could further benefit other tracing tasks for debugging and optimizations.
Bibliography


[18] Laura Carrington, Allan Snavely, Michael Laurenzano, Roy Campbell Jr., and Larry Davis. How well can simple metrics represent the performance of hpc applications? In Supercomputing’05, 2005.


[75] SDSC. URL at [www.sdsc.edu](http://www.sdsc.edu).


