Trace-Based Optimization for Precomputation and Prefetching

by

Madhusudan Raman

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Computer Engineering
University of Toronto

Copyright © 2006 by Madhusudan Raman

Abstract

Trace-Based Optimization for Precomputation and Prefetching

Madhusudan Raman

Master of Applied Science

Graduate Department of Computer Engineering

University of Toronto

2006

Memory latency is an important barrier to performance in computing applications. With the advent of Simultaneous Multithreading, it is now possible to use idle thread contexts to execute code that prefetches data, thereby reducing cache misses and improving performance. TOPP is a system that completely automates the process of detecting delinquent loads, generating prefetch slices and executing prefetch slices in a synchronized manner to achieve speedup by data prefetching. We present a detailed description of the components of TOPP and their interactions. We identify tradeoffs and significant overheads associated with TOPP and the process of prefetching. We evaluate TOPP on memory-intensive benchmarks and demonstrate drastic reductions in cache misses in all tested benchmarks, leading to significant speedups in some cases, and negligible benefits in others.

Acknowledgements

I am greatly indebted to my supervisor, Prof. Michael J. Voss. This work was only completed because of his continued hands-on support and patient guidance.

My parents have been with me, supporting and encouraging me to excel throughout my academic career. I cannot begin to repay what they have invested in me.

I am grateful to Samta for her continued support and encouragement through difficult times. She has given me many reasons to succeed.

I would like to thank Mihai for all his help with Linux, scripting and various ideas, throughout the course of this project. Derek Bruening of MIT has been of invaluable help in the DynamoRio component of my thesis.

Last, but not least, I must mention my friends at the university, who motivated me and accompanied me through the years.

Contents

1 Introduction **1**

1.1 Functional Overview of TOPP 2

1.2 Components of TOPP 3

1.3 TOPP Contributions 4

1.4 Overview of Thesis 4

2 Background and Related Work **6**

2.1 Strata 6

2.1.1 Strata Fragment Builder 7

2.1.2 Strata Trampolines 8

2.2 DynamoRIO 10

2.3 Intel Performance Monitoring 11

2.3.1 PEBS Interrupts 12

2.4 Related Work 13

3 Identifying Delinquent Loads **17**

3.1 Overview of DL Identification System 17

3.2 Kernel Support for Identifying DLs 18

3.3 Coordinating with the User-Level Thread 19

3.4 DL Identification Latency 20

3.5 Overheads and Tradeoffs 22

3.6 DL Identification Summary 25

4 Generating the Prefetch Slice **27**

4.1 Overview of Prefetch Slice Generation 27

4.1.1 Prefetch Slice Requirements 28

4.2 Connected Traces 28

4.3 Generating the Primary DL Loop 30

4.4 Connected Trace Expansion 32

4.5 Aborting CT Generation 33

4.6 Generating the Prefetch Slice from a CT 34

4.6.1 Finding secondary DLs 34

4.6.2 Generating the Basic Block List 36

4.6.3 Finding Basic Block Successors and Predecessors 38

4.6.4 Basic-Block-Specific Reaching Definitions Analysis 38

4.6.5 Criteria for Killing Definitions 40

4.6.6 Generating the Slice 41

Computing Instruction Dependencies 42

Tagging Essential Control Flow Instructions 43

4.6.7 Memory Write Redirection 44

4.6.8 Tracking Jumps 45

4.7 Prefetch Slice Generation Summary 47

5 Executing and Coordinating the Assist Thread **48**

5.1 Overview of Assist Thread Coordination 48

5.2 Synchronization of the Assist Thread 49

5.3 Completing the Prefetch Slice 52

5.4 Invoking Prefetch Slices 54

5.4.1 Prefetch Slice Start Points 55

5.5 Reusing Prefetch Slices 55
5.5.1 DL Patching 57
5.6 Summary of Assist Thread Coordination 57

6 Methodology and Results **58**
6.1 Overview of Methodology and Results 58
6.2 Test System 58
6.3 Evaluation Parameters and Methodology 59
6.4 Benchmarks 59
list 59
array 60
BST 60
mcf 60
6.5 Results 61
6.5.1 TOPP Overheads 61
Overhead due to Strata 61
TOPP overhead 62
Hyperthreading overhead 62
6.5.2 Performance Improvement 64
6.5.3 TOPP Parameters 65
Slack 65
Pause Duration 66
Secondary DL Threshold 66
6.5.4 Cache Performance 67
6.6 Comparison with Hardware Prefetcher 68
6.7 Future Optimizations 69
6.8 Summary 70

7 Conclusion **71**

7.1 Future Work 71

Bibliography **73**

List of Figures

1.1 Hyperthreaded CPU 1

1.2 TOPP Functional Overview 2

2.1 Strata Code Generation and Execution Flow 7

2.2 An example of trampolines 9

2.3 PEBS Buffer and associated data structures 11

2.4 PEBS in action 12

3.1 TOPP Component Overview 17

3.2 Overwriting a DL Instruction With Escape Code 20

3.3 Coordination of User-Level and Kernel-Level Functions 21

3.4 Impact of counter reset value on performance monitoring overhead 22

3.5 Impact on PEBS buffer size on DL Latency 23

3.6 Impact of PEBS buffer size on performance monitoring overhead 24

3.7 Number of DLs versus threshold values for *mcf* 25

4.1 TOPP Component Overview 27

4.2 Connected Trace example - BB's 1,2,4,5,6 form the CT 29

4.3 CT generation and expansion 30

4.4 Primary DL Loop Generation 31

4.5 An example of CT Expansion: (a) The primary DL Loop (BB 1,2,4,5), (b) the expanded CT with an additional basic block (BB 6) 33

4.6 Slice Generation Steps 35
4.7 An example of decoding a fragment 37
4.8 Slicer Loop 41
4.9 An example of essential BBs 44

5.1 TOPP working flowchart 48
5.2 Assist Thread Synchronization Flowchart 50
5.3 Assist thread invoke loop, runs on the second logical CPU 54

6.1 Strata and Slice Generation Overheads 61
6.2 Hyperthreading Overheads 63
6.3 Speedup achieved due to TOPP 64
6.4 Influence of Slack on Speedup due to TOPP 65
6.5 Effect of pause duration on performance 66
6.6 Effect of Secondary DL Threshold on Speedup on *mcf* 67
6.7 Cache Performance of Native Benchmarks 68
6.8 Reduction in cache misses due to TOPP 69
6.9 Speedup due to hardware prefetcher 70

Chapter 1

Introduction

Memory latency is now a dominating factor in many real-world applications. For this reason, cache misses are an important bottleneck in performance-oriented computing [1]. Recent advances in simultaneous multithreading (SMT, also known as hyperthreading [2]) have brought parallel processing capabilities into the desktop PC (Fig. 1.1). However, the vast majority of modern applications are not designed to take advantage of SMT or any sort of parallel processing.

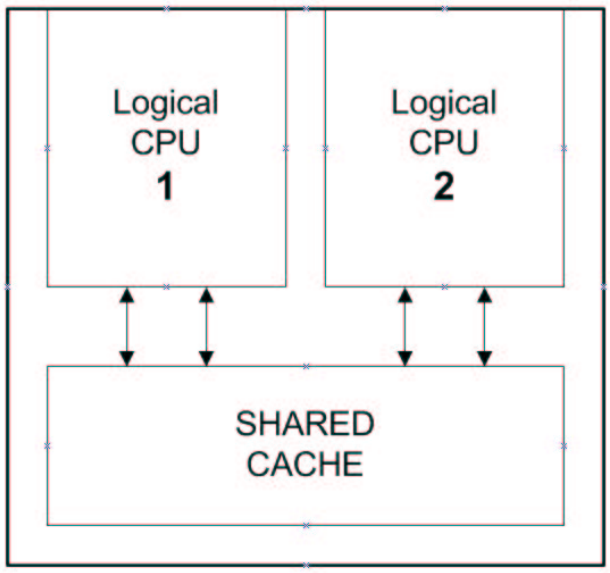


Figure 1.1: Hyperthreaded CPU

In this thesis, we develop and implement a system that addresses the cache miss bottleneck while making use of existing, but underutilized capabilities of SMT CPUs. TOPP (Trace-based Optimization for Precomputation and Prefetching) is a runtime optimization system that dynamically profiles programs, and generates and executes prefetching slices with the purpose of reducing cache misses and speeding up a single-threaded target program.

1.1 Functional Overview of TOPP

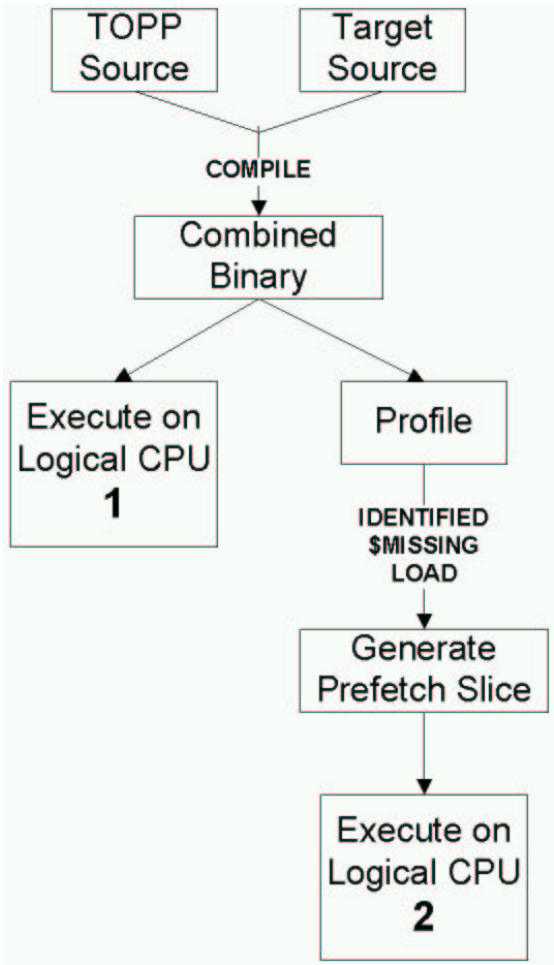


Figure 1.2: TOPP Functional Overview

The fundamental working of TOPP is as follows: TOPP functionality is compiled

into the target program binary. As the target executes, TOPP uses performance monitoring hardware along with statistical sampling to identify load instructions with many cache misses. Once such a load has been identified, TOPP generates a prefetch slice which contains just enough code to compute future memory accesses for the load. As hyperthreaded CPUs have two thread contexts, the prefetch slice is executed on the idle thread context.

As the prefetch slice runs, it executes future iterations of the loop containing the target load and prefetches data into the cache. Since the cache is shared on a hyperthreaded CPU, the target program will not experience cache miss latency while accessing already prefetched data. This results in a reduced cache miss rate and consequently, faster program execution. Fig. 1.2 describes the functionality of TOPP.

Although existing research (Section 2.4) details numerous varied approaches to addressing cache miss latency with multiple threads, ours is the only approach to combine the aspects of dynamic cache miss monitoring, slice generation and prefetching using hyperthreaded CPUs.

1.2 Components of TOPP

TOPP is built as an extension to Strata [3], which is a software dynamic translator (Section 2.1). Strata helps TOPP take control of the target program, interrupt it when a load causing many cache misses is identified, and use the basic blocks around the load to build a prefetching slice. Thus, Strata can be considered the base of TOPP. We also use some library API functionality from DynamoRio [4], which is another software dynamic translator.

To generate a prefetch slice, first a frequently missing load has to be identified. For this, we make use of the Intel Xeon's performance monitoring hardware. We implement functionality in the Linux kernel which allows us to monitor the target program and

obtain samples of instructions causing cache misses. This allows us to keep counts of cache misses caused by specific instructions.

Once such a load has been identified, we use our extensions to Strata to compute the trace that it is on. This trace is usually a loop or set of loops that are involved in computing the load address. We then use the Dynamorio API to generate the prefetch slice itself. Although Dynamorio is also a software dynamic translator, we only use its powerful library for translating and working with machine-level instructions. We use the library API of Dynamorio to prune the trace and generate a minimal prefetch slice.

1.3 TOPP Contributions

In this thesis, we make the following contributions:

- We present the design and implementation of TOPP, a transparent software-only dynamic optimizer that uses runtime profiling to perform cache prefetching at runtime.
- We show our implementation of a runtime profiler using existing performance monitoring hardware on Intel CPUs.
- We demonstrate a novel approach to prefetch slice generation at runtime using Software Dynamic Translators.
- We present performance results of TOPP and show that it can be quite beneficial to cache performance of real-world benchmarks.

1.4 Overview of Thesis

We present a background and describe underlying tools in Chapter 2. In Chapter 3, we describe the process of dynamic profiling and statistical sampling to identify loads

with many cache misses. We then present the process of generating a prefetch slice in Chapter 4. In Chapter 5, we explain how a prefetch slice is executed and coordinated so it can carry out its task of prefetching. Chapter 6 is a description of our evaluation methodology and results. We conclude in Chapter 7, providing a summary of our work and touching upon future research directions.

Chapter 2

Background and Related Work

In this chapter, we introduce systems and algorithms that are used to make TOPP work. We discuss Strata and DynamoRio and touch upon functionality that is used by TOPP. We then describe Intel's performance monitoring system and describe how we use it for our purposes. In the related work section, we discuss related research and contrast TOPP with other existing approaches to data prefetching.

2.1 Strata

Strata is a Software Dynamic Translator. It functions similar to a virtual machine in that it examines and translates instructions before allowing them to execute on the host CPU [3]. Strata's basic function is to enable code control and on-the-fly modification. It accomplishes this by not allowing the target program to execute natively. Instead, native code is copied into Strata's code cache. Only code in the code cache is actually executed on the host CPU. This allows us to take control of executing programs whenever needed, and also allows us to modify code that is about to execute.

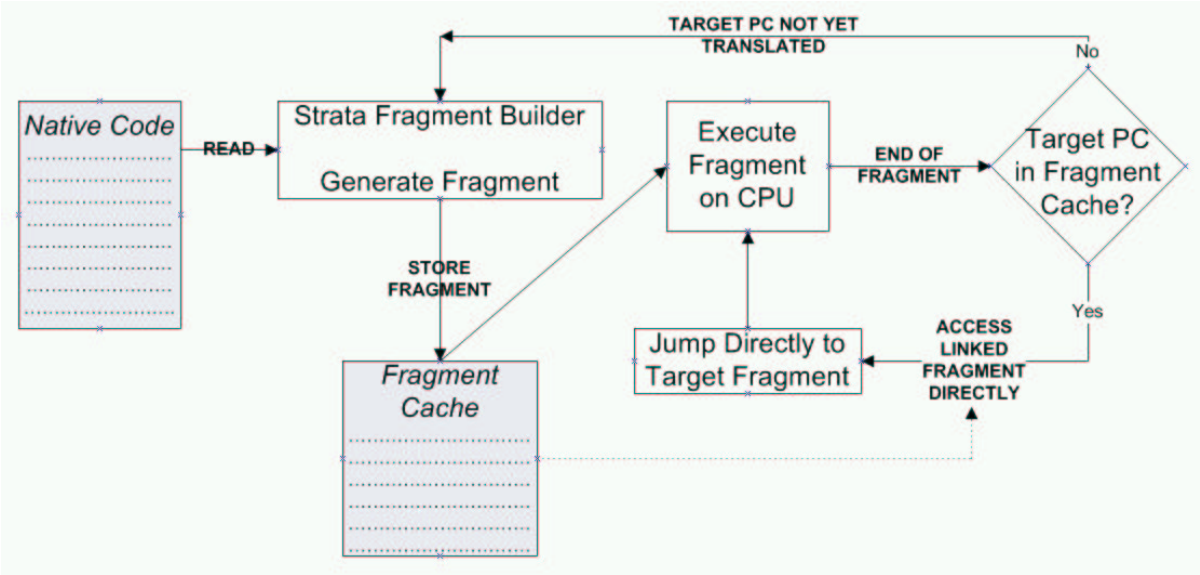


Figure 2.1: Strata Code Generation and Execution Flow

2.1.1 Strata Fragment Builder

Strata uses "fragments" to group contiguously executing instructions. A fragment is the Strata equivalent of a basic block in that control flow can only enter a fragment at the first instruction, and can only leave the fragment at the last instruction.

Strata's fragment builder is invoked by explicit function calls from the target C program (i.e., the program that is to be controlled by Strata). Once the fragment builder has been invoked, it fetches and partially decodes machine code instructions of the target program. Strata only decodes enough of an instruction to compute its length (so that it can fetch the next instruction) and to distinguish between normal instructions and those affecting control flow.

Strata does not interpret or execute single native instructions, instead it natively executes complete fragments only. Therefore, Strata must be able to identify the boundaries of fragments before executing them. To build a fragment, the native instructions to be executed are inspected in sequence. Any instruction that is not a control transfer instruction (CTI) is simply appended to the end of the fragment currently being built.

When a CTI is encountered, one of the following actions is performed:

1. If the CTI is an unconditional jump, Strata continues fetching from the target program counter address (PC). Instructions continue to be added to the current fragment.

2. If the CTI is a conditional jump, Strata ends the fragment and inserts a trampoline (Section 2.1.2).

3. If the CTI is a function call, Strata continues fetching instructions from the callee function and adding them to the current fragment.

4. If Strata encounters a return instruction, it ends the fragment at that point.

Once a fragment is generated, it is stored in Strata's own code cache, called the fragment cache. At the end of the fragment, Strata inserts code that calls the fragment builder with the address of the next native code instruction. Once the fragment is stored, it is executed. Once execution reaches the end of the fragment, the fragment builder is automatically called and begins generating the next fragment, and so on. Figure 2.1 elaborates on this process .

2.1.2 Strata Trampolines

At the time a fragment is being generated by Strata, it has not been executed yet. In addition, since Strata does not interpret machine code, there is no easy way of deducing whether a conditional jump will be taken or not. Therefore, when Strata encounters a conditional jump, it has no way of knowing whether to fetch subsequent instructions from the taken PC or the not taken PC (x86 conditional jumps can only branch two ways). Strata solves this issue by using trampolines and patches.

A trampoline is a section of code (one to three instructions) that mediates a conditional jump and a target. A conditional jump first jumps to a trampoline, which then

decides where execution will continue to. Since conditional jumps have two possible outcomes, there are two trampolines per conditional jump. The conditional jump obviously has to be retargeted (i.e., its target address should be set to the trampoline PC) so that when it is taken, it jumps to the taken trampoline. The not taken trampoline has to be in the fallthrough portion of the conditional jump. Since Strata does not modify native code, the trampoline only exists in the Strata-generated fragment.

```
0x10000 mov ecx -> eax
0x10003 dec eax
0x10005 jnz 11000
0x10009 mov eax,ebx
        ...
        ...
```

a) Native code

```
0x20000 mov ecx -> eax
0x20003 dec eax
0x20005 jnz 20013
0x20009 push 0x10009
0x2000d call <strata>
0x20013 push 0x11000
0x20017 call <strata>
```

b) Unpatched Fragment

```
0x20000 mov ecx -> eax
0x20003 dec eax
0x20005 jnz 20013
0x20009 push 0x10009
0x2000d call <strata>
0x20013 jmp 21000
```

c) Patched Fragment

Figure 2.2: An example of trampolines

The function of a Strata trampoline is as follows. When a fragment is generated, both taken and not taken trampolines consist of the same code - A push of the original (native code) address to start generating from, and a call to the fragment builder. When the fragment is executed and execution reaches either trampoline, Strata begins to generate the missing fragment. Once the fragment is generated, the push and call trampoline is replaced by a direct jump to the newly generated fragment. The next time the trampoline is encountered, execution goes directly to the appropriate fragment.

An example of the above is shown in Fig. 2.2. In this example, Strata starts processing native code from address 0x10000 (Fig. 2.2a). The first Strata fragment is shown in Fig. 2.2b. We see the two trampolines that call the fragment builder with the appropriate address. In this example, we assume that once the fragment is executed, the conditional

jump at 0x20005 is taken. Therefore, we need the fragment corresponding to address 0x11000. This is the second fragment generated by Strata, and for the purposes of this example, we give it a starting address of 0x21000 (fragment not shown). In Fig. 2.2c, we see how the taken trampoline is patched with a jump to the newly generated fragment.

By using trampolines and patching, Strata is able to link fragments together as they are generated. This act of patching jumps so that execution flows directly from one fragment to another is called fragment linking.

2.2 DynamoRIO

The DynamoRio package is similar to Strata in that it is also a Software Dynamic Translator - it also has a code cache in which it executes copied code. In addition, DynamoRio collects and generates traces (Sequences of "hot" basic blocks which repeatedly execute more than a minimum number of times), thus reducing the overhead of decoding and translating code [4]. However, due to the fact that DynamoRio is not open-source, we were unable to use its software translation capabilities for our purposes. This was because TOPP requires major modifications to the basic SDT infrastructure. While DynamoRio provided an API for controlling and instrumenting trace generation, it was not general enough for our needs. We therefore had to use the open-source Strata framework.

DynamoRio has an excellent API that lets us manipulate instructions and instruction lists. The API has many capabilities, such as changing instruction operands, creating instruction lists and encoding them into real machine code instructions into memory, etc. We use DynamoRio as a library for disassembly and instruction manipulation. On the other hand, Strata allows for modification of fragments, but does not provide any support for dissassembly or instruction manipulation. Strata fragments are represented as low-level binary code. For this reason, we make extensive use of the DynamoRio API to perform tasks such as manipulating lists of instructions, computing dependencies, etc.

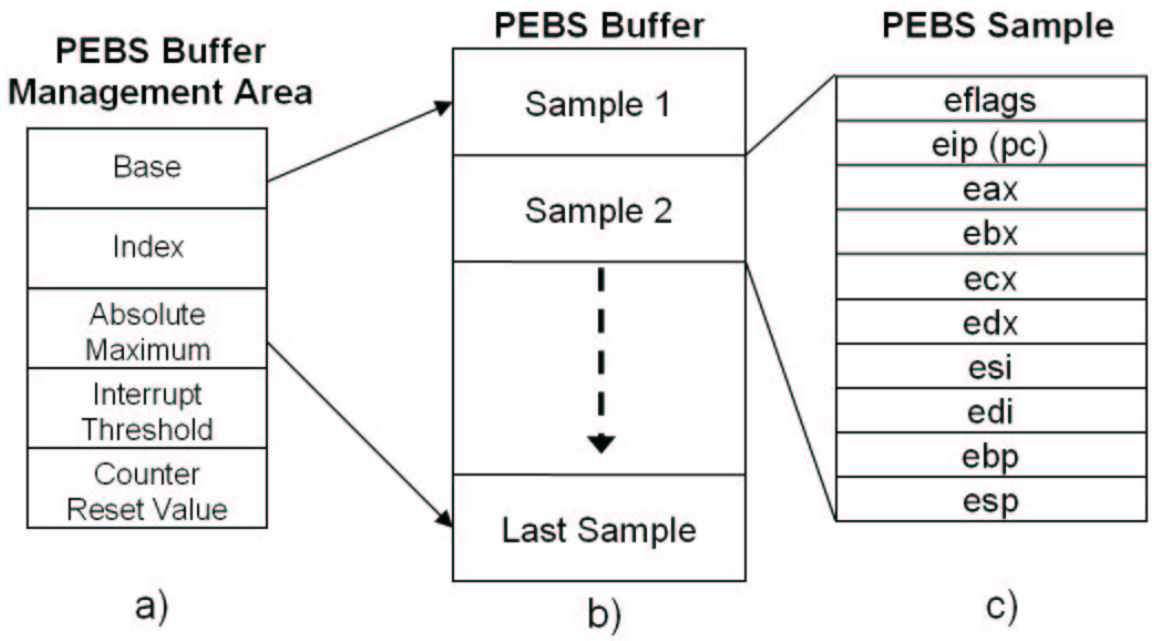


Figure 2.3: PEBS Buffer and associated data structures

2.3 Intel Performance Monitoring

We use the Intel Pentium IV Xeon's existing performance monitoring hardware to dynamically profile the program under analysis. Intel provides a mechanism called Precise Event-Based Sampling (PEBS) [5] to precisely identify the instruction that causes a particular event. We can use this to count cache misses and identify instructions that cause them. The following describes how we use PEBS for our purposes:

1. We initialize a set of registers on the CPU to enable L2 cache miss counting and PEBS.

2. We declare the PEBS buffer (Fig. 2.3b) in kernel memory space. This is where PEBS samples will be stored.

3. We initialize the PEBS buffer management area (Fig. 2.3a). This contains informa-

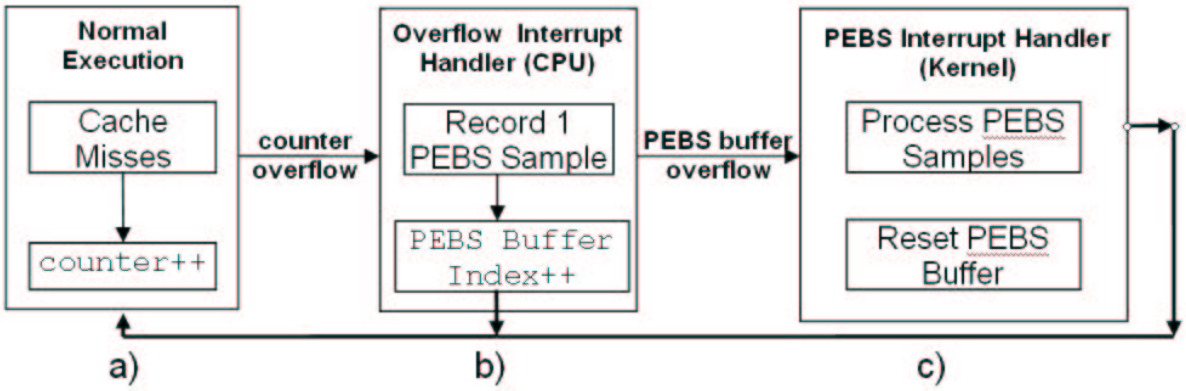


Figure 2.4: PEBS in action

tion on the location and size of the PEBS buffer, and other data such as interrupt threshold and counter reset value.

Performing the above steps will set things up so that PEBS samples can be generated. A PEBS sample (Fig. 2.3c) consists of a snapshot of ten registers that describe the state of the CPU at the time of an event. These are the eight general purpose registers, the flags register and the program counter.

Once the initial setup (steps 1-3, above) is done, PEBS samples are collected and stored in the PEBS buffer automatically by the underlying hardware. This process happens as follows: as we run the program, instructions cause cache misses. Each cache miss causes the counter register to increment (Fig. 2.4a). When the counter register overflows, the CPU looks at the tagged instruction that caused the cache miss. The ten-register PEBS sample is saved to the PEBS buffer (Fig. 2.4b). A pointer in the buffer management area is updated to reflect the position of the latest sample. After the sample is saved, the counter is reinitialized and the CPU resumes normal execution.

2.3.1 PEBS Interrupts

The PEBS Buffer is a limited-size kernel buffer. In our case, it can store a maximum of 100 samples. Since PEBS samples from real-world programs would fill this buffer

many times over, it is necessary to periodically process the samples. The PEBS Buffer Management Area (Fig. 2.3a) includes entries for PEBS Interrupt Threshold, PEBS Index and PEBS Absolute Maximum. The PEBS Index denotes the location of the latest PEBS sample. This is automatically incremented when a sample is added to the PEBS buffer. The PEBS Interrupt Threshold is initialized, in the beginning, to be an address, which is close to the end of the buffer. When the PEBS Index equals the Interrupt Threshold, a PEBS interrupt is generated. In other words, when the PEBS Buffer is almost full, a PEBS interrupt is generated and a user-defined interrupt handler is called. The above mechanism has been implemented by Intel as part of the Pentium IV Xeon's PEBS mechanism. We simply have to define the parameters for interrupt threshold, buffer size, etc.

In experiments performed while evaluating TOPP, the PEBS interrupt is set to trigger when the PEBS buffer is 80% full. This calls a kernel-level routine, which processes all the samples in the PEBS Buffer. The working of the PEBS interrupt handler is described in Figure 2.4c. At the time of initialization, an interrupt handling mechanism is set up. This sets the interrupt handler for a PEBS interrupt to point to our interrupt handler. Each time a PEBS buffer reaches the threshold, a PEBS interrupt is generated, and the interrupt handler is called.

In this way, we use Intel's performance monitoring hardware to generate statistical samples and dynamically profile the target program. (Section 3.2).

2.4 Related Work

Although our approach is unique, many overlapping areas have been covered in recent research.

Simultaneous Multithreading (SMT), or Hyperthreading, was first studied by IBM in 1968 [6]. SMT research perked up in the 1990s, with Tullsen et al simulating an SMT

processor [7]. Since then, many approaches to prefetching with SMT processors have been researched.

Collins et al introduce the concept of speculative precomputation for prefetching delinquent loads using static profiling information with SMT [1]. They define the concept of a "delinquent load" as *static loads that cause the most stalls*. Other approaches using helper threads are found in [8], and [9]. Roth and Sohi present a very fine-grained, instruction-level prefetching approach to reduce cache misses in [10]. Liao et al [11] automate the process of prefetch slice generation using profiling information on an already-compiled binary. Kim and Yeung [12], on the other hand, implement program slicing on source code, thus "improving portability".

Using the first mainstream SMT processor, the Intel Pentium 4, Kim et al provide results from "the first physical experimentation of prefetching using SMT" [13]. In this paper, the authors present a system to generate and execute helper threads for data cache prefetching. They use a profiling tool to identify delinquent loads. Once delinquent loads have been identified, loops around the delinquent load are selected for slicing. The decision of loop selection depth is made using loop counts gathered during profiling. The loops are then sliced using program slicing on source code and code for the assist thread is generated. In addition to the above, this paper also demonstrates the concept of throttling back (i.e., not invoking assist threads for certain periods, depending on dynamic monitoring of shared resource use, cache misses observed, etc. Although this work differs from ours in that the authors perform all DL detection and slicing before compiling the final source code, the concept of throttling the assist thread on an as-needed basis could be an optimization that would enhance our work.

Moshovos et al [14] present slice processors - which completely automate the process of dynamic DL detection and slice generation, all in hardware. The authors add DL identification capabilites to a simulation model of an execution pipeline. Small slices upto 16 instructions long are generated from a buffer of recently committed instructions

using a dynamically generated dependence graph. Typically, these slices are triggered each time a DL is encountered, and execute for the equivalent of a few loop iterations or less. Although the authors do not assume SMT capability, some kind of parallel thread capability is implicit in their test CPU.

In [15], Collins et al present an approach that is similar in purpose to [14]. Delinquent loads are tracked and identified in harware, and prefetch slices generated on demand. The authors also present chaining slices, an approach similar to TOPP, where slices are themselves loops. Synchronization issues and cross-loop dependencies are also taken into account in this work.

Smith introduces a form of prefetching in [16]. The author proposes a CPU architecture with two separate, but interacting processors - one to access memory, and another to execute instructions and perform calculations. One intention of this architecture is to allow the access processor to "run ahead" of the execute processor, thereby achieving data prefetching. The proposed architecture, although not entirely transparent to compilers, allows for significant use of "known compiler and proramming techniques".

Another hardware-based approach is Chou and Shen's "Instruction Path Coprocessors" [17]. In this paper, the authors describe a programmable hardware-only engine which has capabilities such as trace collection and optimization, and stride and linked data structure prefetching. In their implementation, the prefetchers are only invoked when stride or linked data structures are explicitly recognized by specialized hardware.

In contrast to the abovementioned hardware-based approaches, TOPP is a software-only approach that attempts to reproduce the functionality of wholly self-contained and transparent slice processors with real silicon. Ro and Gaudiot describe a hybrid compiler and hardware-driven prefetching mechanism [18].

[19] by Lu et al is a similar work to ours in that it is a software-only approach using a runtime optimizer to detect delinquent loads and prefetch for them. A dynamic optimizer is used to detect phases in programs and build traces for stable phases. Performance

monitoring hardware is used to detect delinquent loads for the currently executing trace. A prefetch instruction is then generated based on the nature of the DL instruction and the prefetch is inserted into the trace. Their work focuses on using a single thread to perform DL detection and prefetching. Although dependencies are calculated for the purpose of generating the prefetch, no prefetch slices are generated in their approach.

Brown et al describe a system for prefetching using chip multiprocessors (CMPs) in [20], using a simulation environment. Another work with considerable overlap with ours is Lu et al's [21]. In this work, a dynamic optimizer is used with a CMP to detect phases, generate traces and detect DLs on loops. The authors also implement prefetch slice generation and synchronization of main and helper threads. This work differs from TOPP in that some profiling information (for example, branch hints) is used. Furthermore, the target CPU for this paper is a CMP, which considerably reduces the overheads of dynamic optimization and prefetch thread execution as compared to an SMT CPU. Another distinguishing difference is their test processor is a RISC Sun UltraSparc CPU whereas we focus on the CISC Intel Pentium Xeon, which has unique issues with dynamic optimization.

Having reviewed existing research in this area, it is our opinion that ours is the only approach so far to combine the concepts of dynamic DL detection, runtime prefetch slice generation and prefetching using hyperthreading.

Chapter 3

Identifying Delinquent Loads

3.1 Overview of DL Identification System

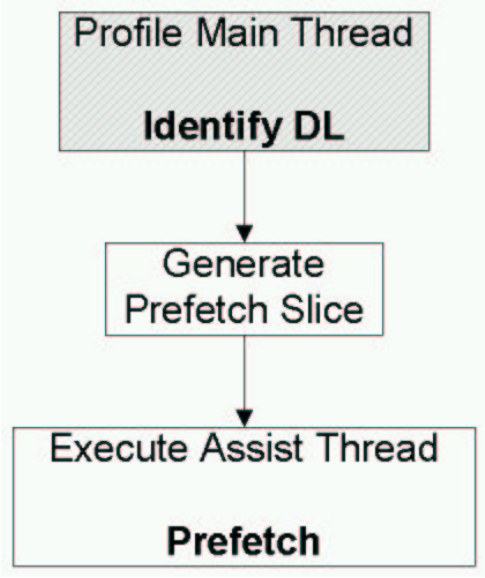


Figure 3.1: TOPP Component Overview

In most programs, a small number of loads are responsible for most stalls due to cache misses. These are defined as "delinquent loads" [1]. In our implementation of TOPP, we are not able to perform statistical sampling by measuring stalls caused by instructions. However, we are able to count cache misses caused by instructions. Therefore, for the

purposes of our work, delinquent loads are considered to be those that cause a number of cache misses, above a preset threshold. In this chapter, we describe how we identify such loads.

The delinquent load (DL) identifier is the first module in our system (Fig. 3.1). The purpose of this module is to dynamically profile the main program and keep count of instructions that cause the most cache misses. We make extensive use of the Pentium Xeon's performance monitoring hardware to perform dynamic profiling. Kernel-level data structures are used to store sample counts, and a signal is sent to userspace whenever a delinquent load is identified.

In the following sections, we first describe how we use existing performance monitoring hardware to dynamically collect profiling information. Then, we describe how we use this information to identify delinquent loads. We then discuss the action taken to interrupt the user-level process when a delinquent load is identified. We also discuss parameters affecting overhead and accuracy of the dynamic profiling process.

3.2 Kernel Support for Identifying DLs

This section describes the kernel-level support implementation for processing PEBS samples and using them to identify delinquent loads. As described in Section 2.3, we setup and use Intel's performance monitoring hardware to collect PEBS samples and generate interrupts when a number (80, in our implementation) of samples have been collected. We have implemented a kernel-level interrupt handler to process PEBS samples. This interrupt handler processes PEBS samples in the following way:

1. Each sample's program counter value (Fig. 2.3c) is read and an entry created in a hash table. The purpose of the hash table is to keep count of how many samples we have seen with a particular program counter address (i.e., a particular instruction). Thus, if the entry exists in the hash table, its count is incremented. All samples in

the buffer are processed in this manner.

2. The buffer is reset, i.e., the PEBS Index is reset back to the base.

3. Once an entry's count reaches a certain threshold, the associated instruction is deemed to be a delinquent load.

Once the kernel-level handler identifies a delinquent load, a signal is sent to the user-level process, to interrupt it. The program counter address of the delinquent load instruction is made available to userspace, so that a prefetching slice can be generated for it.

3.3 Coordinating with the User-Level Thread

As a delinquent load is identified by the kernel-level module, this has to be communicated to the user-level program. The user-level program needs to be interrupted so that a prefetch slice can be generated and started up. This is done as follows: Earlier on, as part of the PEBS initialization process, a custom signal is defined and a user-level signal handler is set up to wait for this signal. In the kernel-level module, when a delinquent load is identified, a signal is sent to the user-level process (Fig. 2.4c). The user-level process, when interrupted, runs a signal handler.

The signal handler uses a system call to get the program counter address of the delinquent load (Fig. 3.3). The purpose of the signal handler is to ensure that the main program is interrupted and the prefetch slice generator is called. We have chosen to accomplish this by overwriting an instruction at or near the delinquent load with a *call* instruction that calls the Strata fragment builder. Figure 3.2 is an example of how code around the DL is changed to insert escape code.

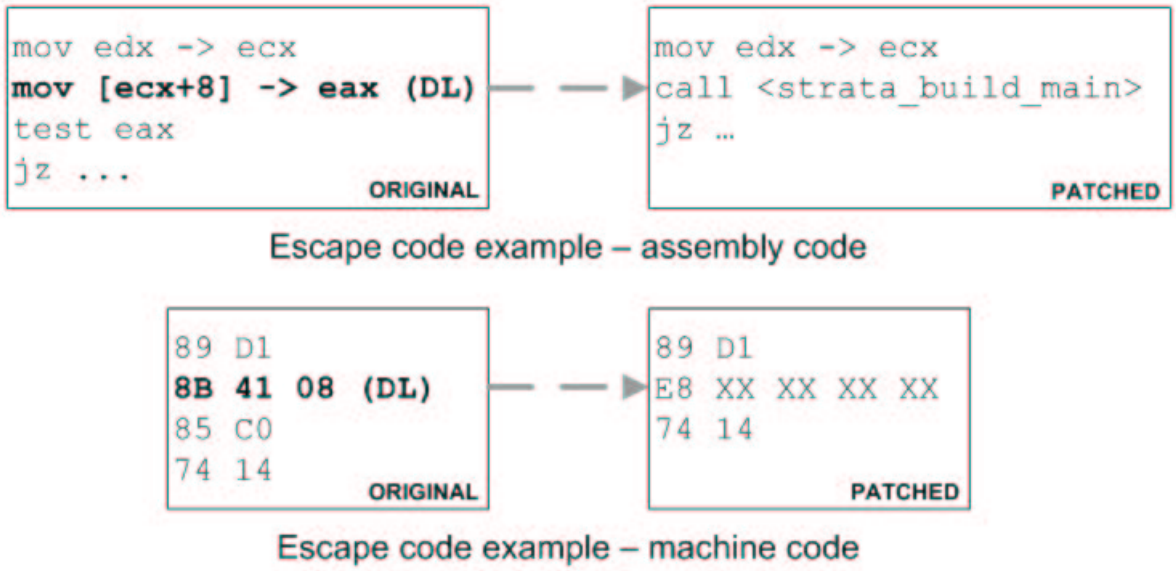


Figure 3.2: Overwriting a DL Instruction With Escape Code

3.4 DL Identification Latency

DL Identification Latency refers to the delay between a load *becoming* delinquent and the point at which it is *identified*. This latency exists because the PEBS buffer stores a few tens of samples before processing them, thus, execution could have moved on from the loop in which the delinquent load was. Overwriting the delinquent load instruction is still correct because execution might (and, in our experience, does) return to the delinquent load loop, at which point our prefetch slice can get into action. However, as a precaution, we continue performance monitoring and DL identification even after we overwrite the DL instruction with escape code. This is done so that, if execution does move on from this DL loop, other DLs in other loops have a chance to be identified. If that happens, the newly identified DL is marked for processing, the old overwritten instruction is restored, and the new DL instruction is overwritten with escape code. Figure 3.3 describes this procedure.

Another issue caused by DL identification latency is that sometimes, the current program counter is too close to the delinquent load. The call instruction we use to

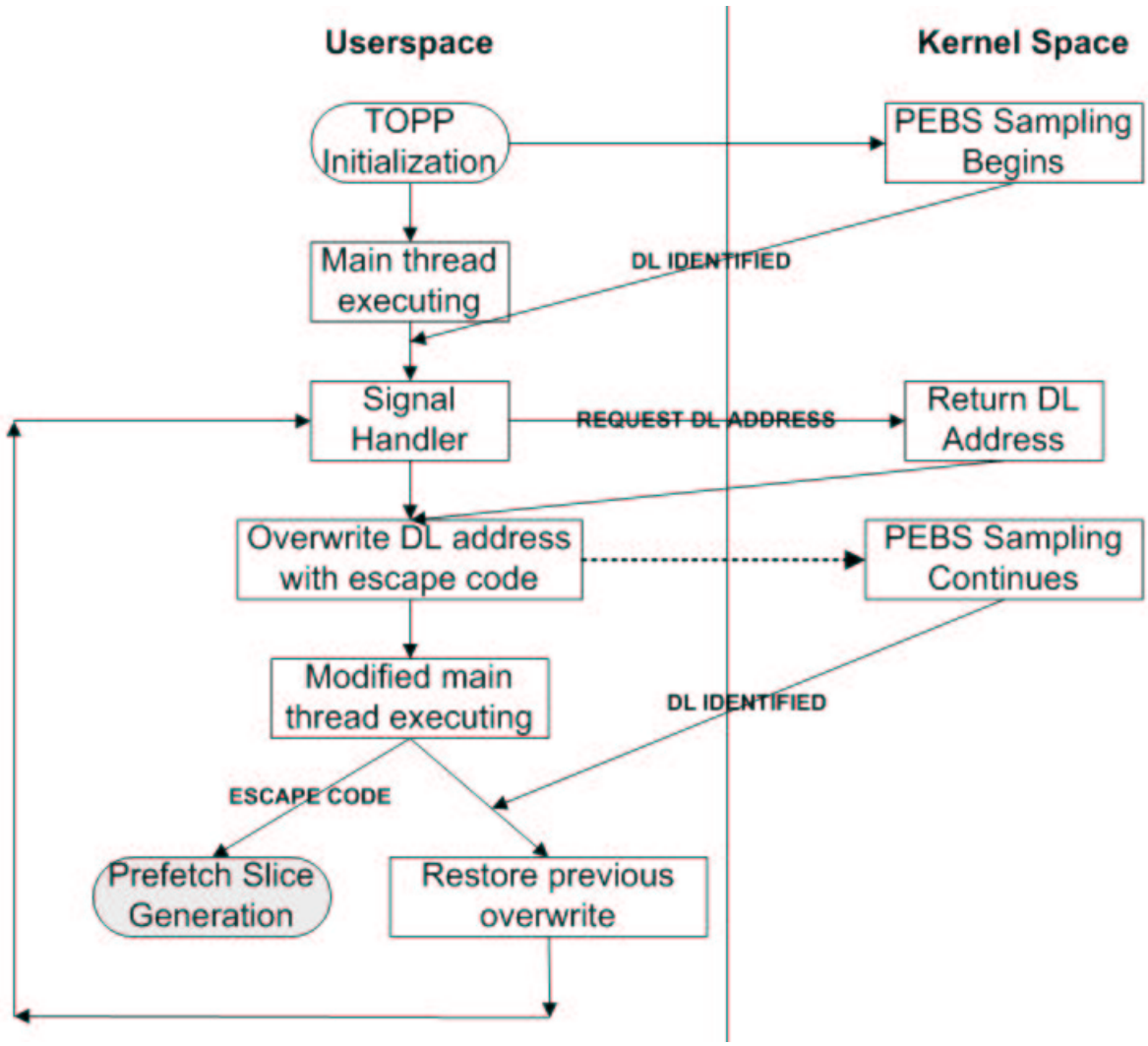


Figure 3.3: Coordination of User-Level and Kernel-Level Functions

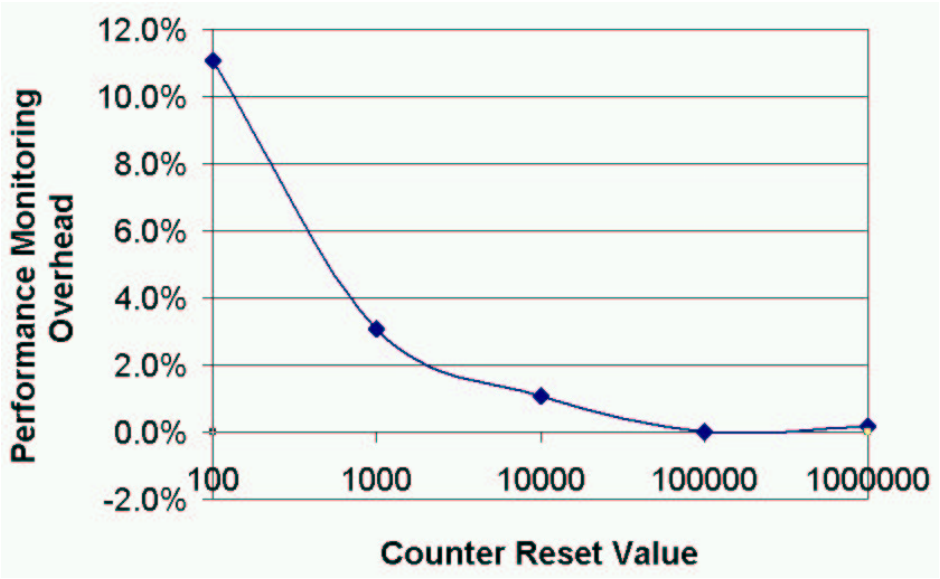


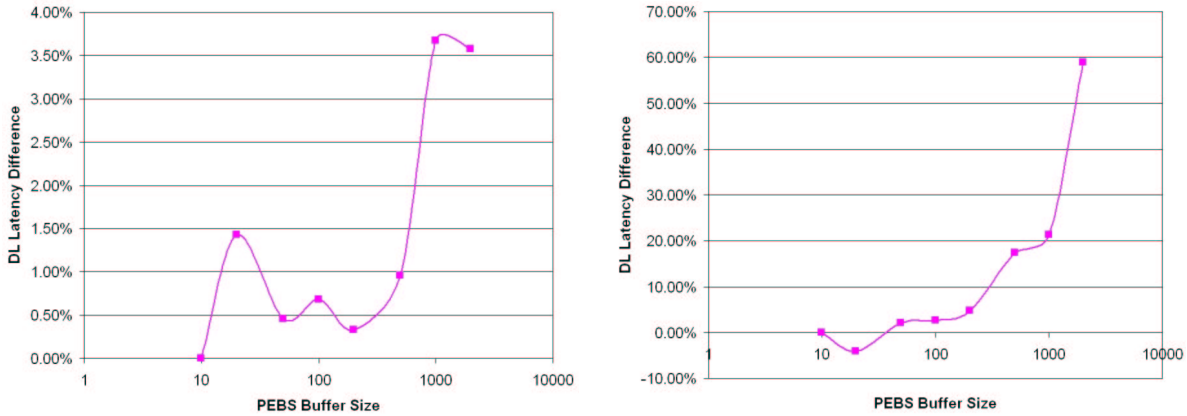
Figure 3.4: Impact of counter reset value on performance monitoring overhead

escape is five bytes long. If, for example, the delinquent load instruction is three bytes long, and the program counter is at the next instruction, overwriting the DL instruction will, in fact,overwrite the instruction about to be executed, and cause an error. We avoid this by checking the return address of the signal handler. Thus, if there is not enough room between the DL and the return address, we overwrite the return address itself. As, by definition, this is very close to the DL instruction, we are guaranteed, in practice, that execution will remain on the delinquent load loop.

After overwriting the appropriate instruction, the signal handler is allowed to return. The next time execution reaches our overwritten code, the Strata fragment builder is invoked. The overwritten code is immediately restored by the fragment builder, which then commences the task of prefetch slice generation (Chapter 4).

3.5 Overheads and Tradeoffs

In our design of DL identification, there are three different variable parameters, each with its own effects:



a) *mcf* b) *list*

Figure 3.5: Impact on PEBS buffer size on DL Latency

Counter reset value - This is the number of cache misses for which the event counter will overflow, generating a PEBS sample. The higher this number, the fewer PEBS samples we have, and consequently, the lower "resolution" we have. On the other hand, the lower the number, the more PEBS samples are generated per unit time, and the higher the performance monitoring overhead is. Fig. 3.4 shows the performance monitoring overhead for different values of counter reset as measured on our test system using the *mcf* benchmark. Overhead is calculated as percentage slowdown seen in runs with performance monitoring turned on versus those with performance monitoring turned off.

PEBS Buffer size - This affects the number of samples stored in the buffer before they are processed. Increasing this number reduces the frequency of calls to the kernel-level PEBS sample processing functions. However, increasing the buffer size also affects when a delinquent load is caught. For example, if the buffer is too large, we may have an unnecessarily long wait between the time a delinquent load becomes "hot" and the time it is identified. In other words, a large buffer size increases the latency between the time a load crosses the threshold of becoming delinquent and the time it is identified as such. Experimentation has shown that in some cases,

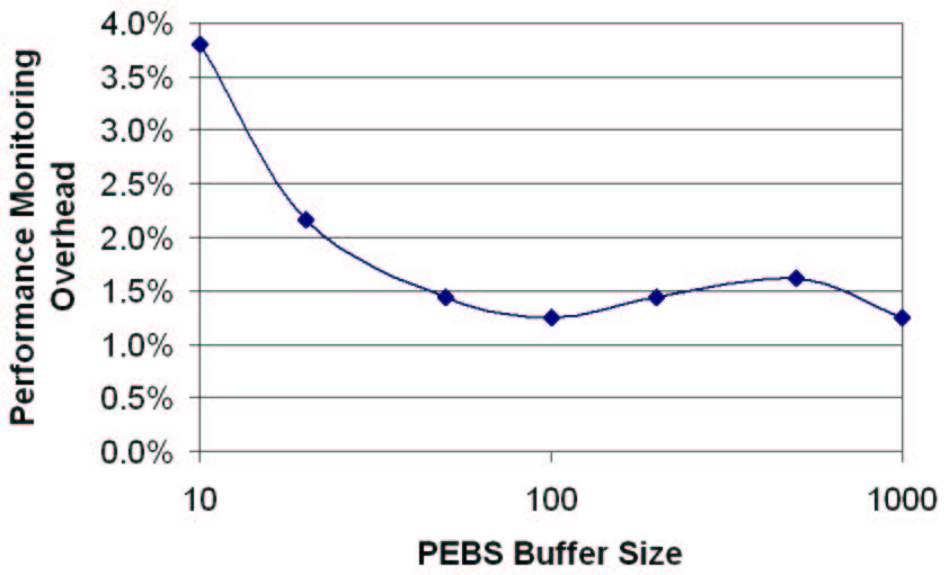


Figure 3.6: Impact of PEBS buffer size on performance monitoring overhead

the program actually runs past the end of the loop containing the delinquent load, and only then is the DL identified as such.

Long DL latency has an effect on the efficacy of the DL identification system as prefetch slice generation, and by extension, prefetching, are directly triggered by DL identification. For this reason, the sooner a DL is identified, the earlier TOPP can start the assist thread to prefetch for it. We ran experiments on two benchmarks - *mcf* and *list*, varying PEBS Buffer size while keeping the counter reset value and DL threshold constant. We normalize latency calculations with respect to readings from the smallest PEBS Buffer size of 10 entries. From Fig. 3.5a and Fig. 3.5b, we see that increasing PEBS buffer size generally causes an increase in DL latency, although the impact of this can vary considerably across benchmarks. Figure 3.6 shows that PEBS overhead mostly decreases as PEBS Buffer size is increased.

DL threshold - This is a parameter that applies to the kernel-level hash table counts (Section 3.2). The lower the DL threshold, the sooner an instruction is identified as a delinquent load. However, too low a threshold causes too many loads to be

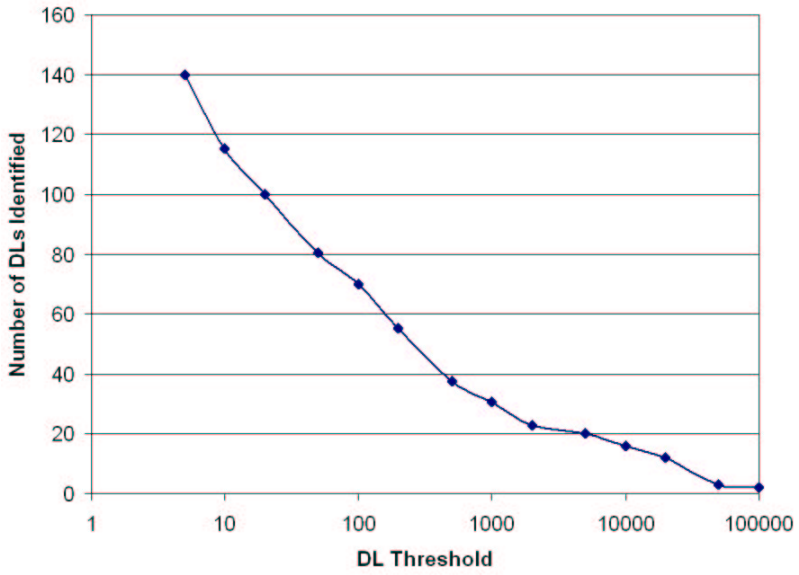


Figure 3.7: Number of DLs versus threshold values for *mcf*

identified as delinquent. This leads us to generate prefetching slices for many traces that do not execute long enough to amortize the associated overhead. Limiting slice generation and prefetching to loads that have high sample counts thus makes sense. Fig. 3.7 shows the number of DL samples measured in a benchmark above various thresholds.

From the above analysis, we are able to choose concrete values for the parameters. We choose a counter reset value of 10000 as it results in quite low overhead, alongwith a reasonable DL latency. For the same reason, we choose a PEBS buffer size of 100 entries. We choose a DL threshold size of 500 to keep the number of different slices generated to a manageable amount.

3.6 DL Identification Summary

In this chapter, we have described how a delinquent load is identified. We have introduced performance monitoring and Precise Event-Based Sampling, and have described how we use them to identify delinquent loads. We have shown how the DL identification module

interacts with the running program and how it triggers prefetch slice generation. We have also discussed issues relating to the overhead and efficiency of DL identification. In the next chapter, we will describe the process of prefetch slice generation that is triggered by DL identification.

Chapter 4

Generating the Prefetch Slice

4.1 Overview of Prefetch Slice Generation

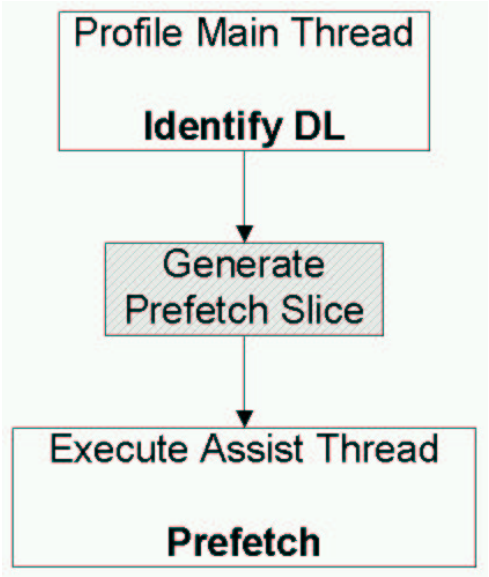


Figure 4.1: TOPP Component Overview

Generating the prefetch slice given only the delinquent load address is a multi-step process that is at the core of TOPP. It consists of two steps - first, the trace of execution that contains the delinquent load is computed. Second, this trace is pruned to generate a slice, which is the smallest set of instructions sufficient to prefetch the delinquent load.

In this chapter, these two processes and associated tasks are described in detail.

4.1.1 Prefetch Slice Requirements

The objective of a prefetching slice is to prefetch for the main thread. To effectively prefetch, the slice needs to run alongside and slightly ahead of the main thread. Since delinquent loads are found in loops, the prefetch slice must exit its version of loop at the same time the main thread exits. To successfully prefetch, the slice must contain enough instructions to completely and correctly compute successive delinquent load addresses.

The above are criteria for a correct prefetch slice. In addition, a prefetch slice should run for at least a minimum amount of time. This is because of the overhead associated with invoking the prefetch slice - if there are too many invokes over the course of a program, we may lose any benefit gained from prefetching. To this end, the prefetch slice must contain, if necessary, some auxiliary fragments that are not necessary to compute the delinquent load, but affect whether execution "falls off" the slice.

In the remainder of this chapter, we will describe how we generate a prefetch slice that satisfies the above criteria.

4.2 Connected Traces

We define a "connected trace" (CT) to be a set of execution paths that the delinquent load lies on. A delinquent load, is executed repeatedly (since it has multiple cache miss samples), therefore it must lie on a loop. However, depending on the circumstances, the delinquent load may lie on an inner nested loop and may have complicated control flow around it. Although many execution paths may contain the delinquent load, a connected trace is computed such that it includes frequently taken execution paths. Thus, the CT need not include all execution paths that the DL may lie on.

In other words, a CT may be defined as *basic blocks comprising a set of frequently*

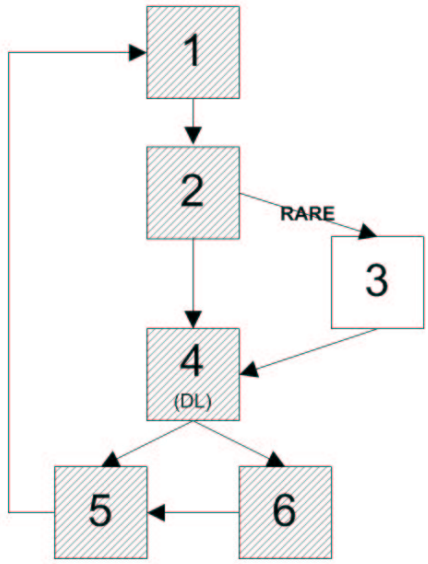


Figure 4.2: Connected Trace example - BB's 1,2,4,5,6 form the CT

occuring execution paths that contain the DL, constituting at least one loop around the DL.

We build the connected trace first, to obtain the set of fragments around the DL, after which we construct the prefetch slice. In Section 4.1.1, we determined that a prefetch slice must run for a minimum amount of time. It is with this requirement in mind that we generate a "connected trace" rather than a single trace or loop. Thus, a connected trace is the set of fragments that includes the delinquent load, and is expected to execute for a significant amount of time. We will define "significant" in section 4.4. Figure 4.2 is an example of a connected trace.

Our methodology for generating a CT is as follows: First, we generate the main loop that contains the delinquent load. Then, if necessary, we expand the CT, one fragment at a time, in an iterative process (Fig. 4.3). The following sections describe this procedure.

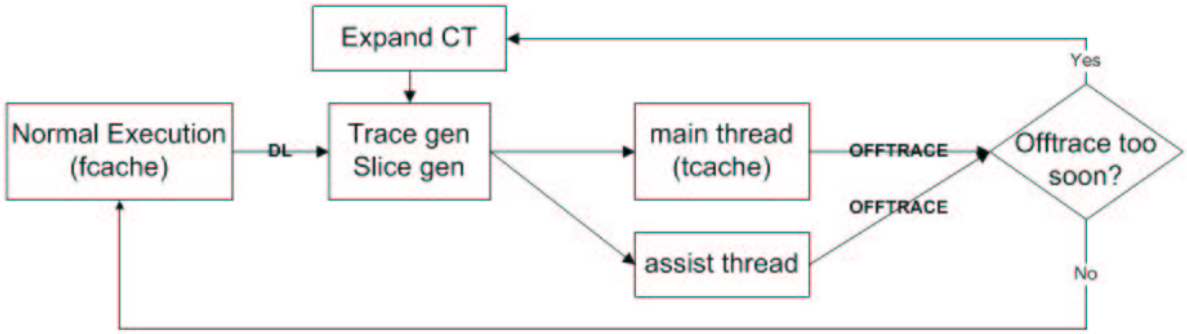


Figure 4.3: CT generation and expansion

4.3 Generating the Primary DL Loop

The primary loop is generated as soon as we identify the delinquent load. The strata fragment builder is called by the user-space signal handler (Section 3.3). At this point, the delinquent load address and the address of the last executed instruction are made available to the fragment builder.

Once the fragment builder has been called with a delinquent load, its objective is to store loop fragments in a "trace cache", just like Strata stores fragments in the fragment cache. Fragments in the trace cache (tcache) correspond to fragments in the fragment cache (fcache) analogous to how fragments in the fragment cache correspond to native code. Because of the analogy, we are able reuse the fragment builder to build CTs.

To generate the first fragment, given the DL, we look up the fcache fragment that contains the DL. We then allocate a region of memory for the trace cache. The Strata fetch/decode/translate loop is then executed. This loop copies over instructions from the fcache to the tcache, stopping at a conditional jump or return instruction (essentially, from where control flow cannot be predicted without executing the fragment). This ends generation of the first trace fragment. As the fragment is generated, when we encounter the last executed fcache instruction (the one after which we jumped into trace generation), we note its corresponding tcache instruction address. After the first fragment is generated, we resume execution in the trace cache starting at this particular address.

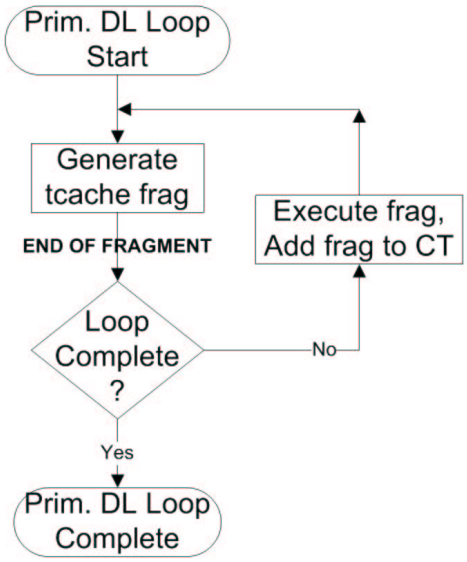


Figure 4.4: Primary DL Loop Generation

This step maintains program correctness and ensures that instructions are executed in the correct sequence. In this way, we seamlessly transfer program execution from the fragment cache to the trace cache.

The first CT fragment ends with two "trampolines" (Section 2.1.2) because no other fragments have yet been copied from the fragment cache into the trace cache. The function of the trampoline is to call the fragment builder with the starting address of the fragment to be executed next, thereby starting the process of fragment linking (Section 2.1.2) Unlike normal strata execution however, we only link fragments that are on the same CT, and explicitly avoid creating inter-CT links. This helps us keep track of whether execution stays on a particular CT.

We continue generating and linking fragments until we find a jump instruction whose target address is the beginning of the first fragment. At this point, the primary DL loop is complete (Fig. 4.4). The prefetch slice can now be generated and started. In some cases, the primary DL loop represents the entire connected trace. However, in many cases, more fragments need to be added to form a "complete" CT. This is described in the following section. In some rare cases, a required fragment is not in the fragment

cache - i.e., it has not been translated by Strata yet. In this case, CT generation must abort and return to normal fcache mode, as CT generation in its current form can only operate on fragments that are already in the fragment cache.

Once the CT has been generated, the prefetch slice is built (Section 4.6). At this point, the prefetch slice is executed as the assist thread, and the CT as the main thread. Executing the CT as the main thread preserves correctness, since CT fragments are exact copies of Strata fcache fragments. Since the CT only has a limited number of fragments, it eventually encounters an unlinked trampoline - i.e., control flow needs to go to a fragment that is not part of the CT. At this point, execution reenters the fragment builder. This constitutes going "offtrace", at which point the prefetch slice is also stopped and normal mode is reentered.

When a CT goes offtrace, we count the number of times the DL has been executed using previously inserted instrumentation. If this count is above a certain threshold, the CT is deemed to be run long enough, and the CT is "committed" - i.e., it and its fragments are flagged as having run long enough. Committing a CT fragment protects it from being discarded in the future (Section 4.5). On the other hand, if the count is below the threshold, the CT is too short, and needs to be expanded. CT expansion is discussed in the following section.

4.4 Connected Trace Expansion

Expansion of the trace is a simple procedure. We generate only the fragment that was missing and link it with the CT, by patching the trampoline. We then generate a new prefetch slice and execute it. If we need to expand again, we repeat the above steps as long as is needed. Trace expansion is thus an iterative process. In practice, no more than a few expansions are required to generate a CT that executes the DL thousands of times and satisfies our amortization requirements.

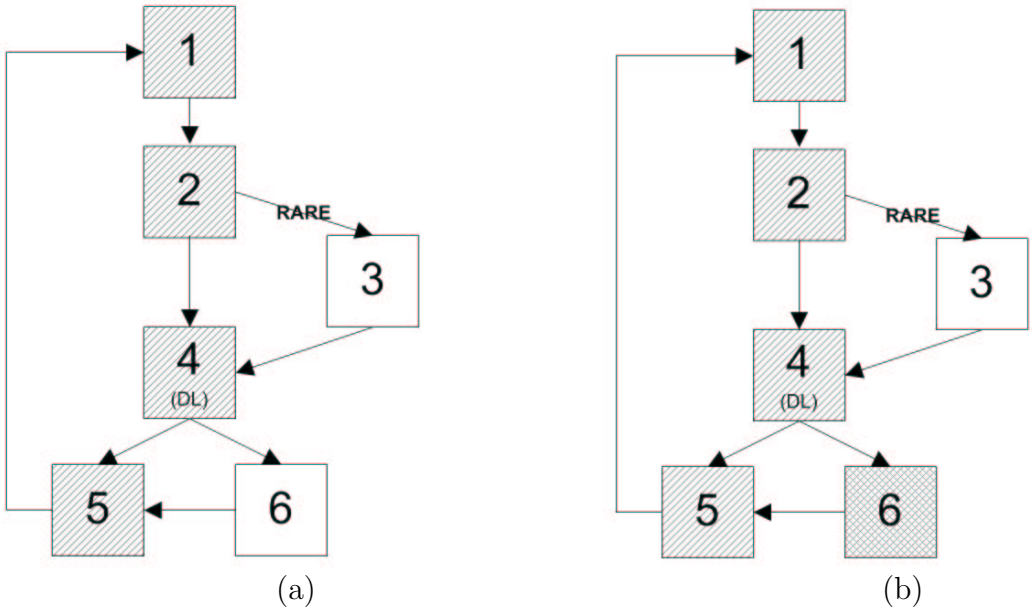


Figure 4.5: An example of CT Expansion: (a) The primary DL Loop (BB 1,2,4,5), (b) the expanded CT with an additional basic block (BB 6)

Once a CT executes above the threshold of iterations, it does not need to expand any more. It and all its constituent fragments are flagged as "committed". However, this does not preclude future expansion. If future invocations of the CT do not execute sufficiently long enough, the CT will be expanded again.

Each time the CT is expanded, we generate a new prefetch slice for the expanded CT. Execution of the prefetch slice follows immediately. If further expansions are needed, the process is repeated.

4.5 Aborting CT Generation

Sometimes, we need to abort generating a CT. For example, we might not complete a loop after adding hundreds of fragments. In expansion mode, we may reach the end of a function, and process a return fragment. Or, we may encounter a fragment that hasn't been translated by Strata from native code yet (i.e., The fragment does not even exist in Strata's fragment cache). At instances like these, we need to abort CT generation, either

partially or completely.

Aborting a CT, however, is not completely straightforward. In many cases, part of the CT is valid, and does not have to be thrown away, whereas other parts need to be thrown away. For example, if we commit a DL loop, but run across an aborting condition in expansion mode, we do not want to discard the entire CT, just the fragments added in expansion mode.

When a trace is aborted, all uncommitted fragments are deleted. In addition, all patches used to link fragments are recorded in the process of trace generation. At the time of aborting, this information is used to undo the trampoline links from committed fragments to fragments that are to be deleted. Thus, all fragments in the CT look exactly like they did at the time of the last commit. If we are aborting an entirely uncommitted CT (i.e., a new CT), we also delete its corresponding prefetch slice and remove other related information about it so that we can start from a clean slate the next time the DL is encountered.

As a final step, to resume normal operation, we set the builder mode to regular fragment generation, resume DL detection and begin executing fragments from the fcache.

4.6 Generating the Prefetch Slice from a CT

Once a CT has been completed, we generate the prefetch slice using a process of slicing. We use the dynamorio suite (Section 2.2) to work with instructions, modify operands, and encode the prefetch slice in memory. In the following subsections, we describe the steps involved in generating the prefetch slice.

4.6.1 Finding secondary DLs

In some CTs, we have found that more than one instruction shows a large number of cache misses. Due to the fact that we use a threshold system to detect cache misses, we

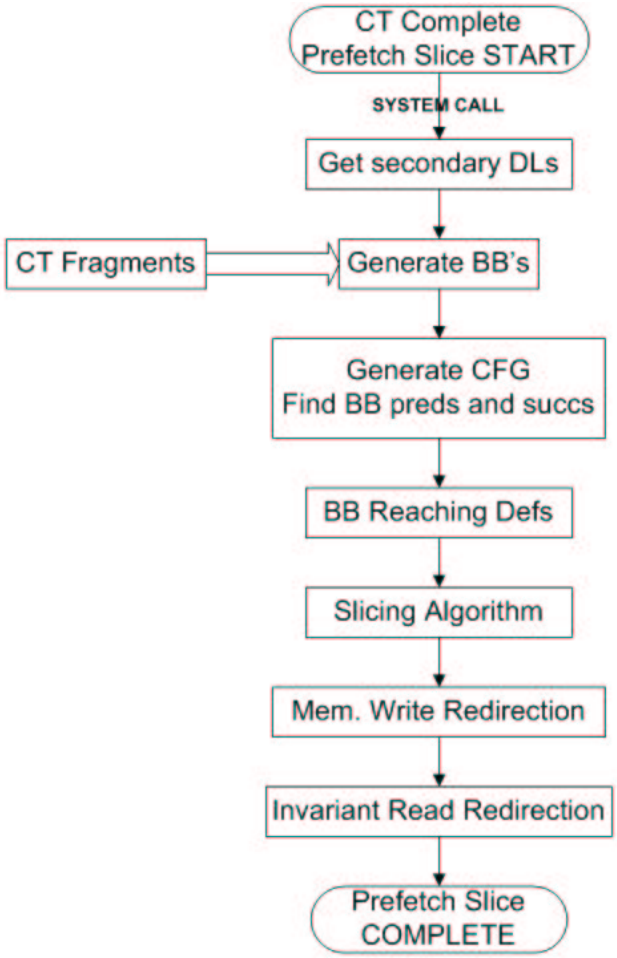


Figure 4.6: Slice Generation Steps

only have one "primary" delinquent load for a CT. We have implemented a system for identifying "secondary" delinquent loads.

A secondary DL has two characteristics - it is on the same CT as the primary DL, and it has a significant number of cache misses. Since we use sampling to identify DLs, the second characteristic reduces to a requirement for a significant number of samples in the kernel-level hash table (Section 3.2). We introduce a secondary DL threshold, which is a fraction (usually half) of the primary DL threshold.

Just after generating the CT with the primary DL, but before beginning the process of slicing, a system call is made. A list of instructions in the CT is passed to the kernel, in the form of fragment cache addresses. The kernel-level function looks up each address

in the DL sample hash table, and prunes the list, leaving only those addresses that have sample counts greater than the secondary threshold. It then copies over the pruned list back to userspace. This provides us with a listing of primary and secondary DLs.

Back in userspace, the instruction addresses are translated from fragment cache to trace cache addresses, using a 1-to-1 mapping (which is generated by the fragment builder during CT generation). This list of delinquent loads is then passed along for slicing. The slicing process makes no distinction between primary and secondary DLs, and generates a slice which prefetches for all DLs.

There are some factors in choosing an appropriate secondary threshold. Too high a threshold causes too few secondary DLs to be identified, thus missing some DLs. Too low a threshold could include too many DLs, thus bloating the prefetch slice and possibly slowing it down so much that it cannot run ahead of the main thread. In addition, some load instructions are related (e.g., Instructions operating on different fields of a single data structure), and prefetching for one also prefetches for the other. In cases like this, prefetching too many DLs could possibly be redundant. We have selected a secondary threshold of 50% of the primary DL threshold based on experimental observations (Section 6.5.3).

4.6.2 Generating the Basic Block List

This is the first step of the slicing process. DynamoRio has its own format for instructions, instruction lists, etc. To use dynamorio on a CT that has been generated by Strata, the CT has to be translated into DynamoRio's instruction format. To this end, we use a custom data structure that corresponds to each CT fragment. Each *basic_block* structure stores the following information:

- Starting and ending PC of the CT fragment that corresponds to the basic block
- A list of all the instructions that belong to the basic block.

- Other data structures that help in dataflow and control flow analysis - e.g., A list of successors, predecessors, the reaching definitions in-set, etc.

For each CT fragment, we "decode" each instruction into dynamorio's *Instr* format and add it to the basic block - specific instruction list. If we come across a delinquent load address, we note its location by using a pointer to the *Instr*. Normal instruction processing continues until we reach a trampoline. If the trampoline is patched (i.e., it is a jump to the target fragment), we process and add it, and then skip to the next trampoline, if it exists. If, on the other hand, we encounter an unpatched trampoline, it means that the target fragment is not part of the CT. In this case, we add a dummy jump to the instruction list to denote an offtrace condition.

Once all fragments have been processed, we have a list of basic block structures, each containing a list of instructions in dynamorio format.

```
464EA077  mov edx,[ecx+0x10]
464EA07A  test edx,edx
464EA07C  mov ecx,edx
464EA07E  jnz 0x464ea094
464EA080  mov dword [0x8160d1c],0x0
464EA08A  push dword 0x4000d27b
464EA08F  jmp 0x805eafa
464EA094  jmp 0x464ea0a8
```

a. Input fragment

```
0x464ea077 mov    0x10(%ecx) -> %edx
0x464ea07a test   %edx %edx
0x464ea07c mov    %edx -> %ecx
0x464ea07e jnz    $0x464ea094
0x00000000 jmp    instr_ptr
TARGET OF CTI:  0x00000000 nop
0x464ea094 jmp    $0x464ea0a8
```

b. BB Instruction List

Figure 4.7: An example of decoding a fragment

In Fig 4.7, the CT fragment is shown on the left. As we can see, the conditional branch (at 0x464ea07E) leads to a trampoline. The fallthough part of the trampoline (0x464ea080-0x464ea08f) has not been patched, thus it is an offtrace condition. The taken part (at 0x464ea094) has been patched, and jumps to the next CT fragment. When the

fragment is decoded into a basic block instruction list in dynamorio, the offtrace part is replaced by a jump to a dummy instruction.

4.6.3 Finding Basic Block Successors and Predecessors

Once we have generated the basic block list, we need to deduce control flow information before proceeding further with dataflow analysis. Specifically, each basic block needs to be "connected", i.e., its predecessors and successors need to be found. We implement this by first generating a control flow graph, and then using it to populate the basic block list with predecessors and successors.

To generate the control flow graph, we first add empty basic block data structures representing the entry and exit basic blocks. We then iterate through each basic block to find its successor basic blocks. Because of the nature of strata trampolines, and the way we process them (Section 4.6.2), the last one or two instructions in a basic block are the jumps that tell us where control is transferred to. Thus, to find the basic block that corresponds to a particular jump, we simply look for the basic block whose starting PC is the target address of the jump in question. In this way, we are able to generate the edges of the control flow graph. Some jumps may represent offtrace conditions. We denote these by adding an edge with the exit basic block as the target.

Thus, we now have a control flow graph, with serial numbers corresponding to each basic block as the nodes, and directed edges representing jumps between basic blocks. We then iterate through all the edges, and add predecessors and successors to each basic block structure. At the end of this process, each basic block has a list of predecessors and a list of successors.

4.6.4 Basic-Block-Specific Reaching Definitions Analysis

We perform two reaching definitions analyses to find dependent instructions so we can generate the prefetch slice. The first is a global, basic-block-at-a-time analysis. The

second is a local, instruction-specific analysis, which builds on the global reaching definitions result to generate instruction-specific reaching definition lists. In this section, we describe the steps involved in performing the global analysis.

The first step in a reaching definitions analysis is generating gen and kill sets for each basic block. Here, our approach differs from a standard compiler-level reaching definitions analysis in that we work with registers and memory operands rather than the usual variables and constants.

We populate gen sets one basic block at a time. We iterate through the instruction list of a BB, adding each destination operand to a global list of definitions (GDL). Any register or memory operand that is written to is considered a definition and included in the GDL. At the same time, we add the definition to the BB gen set. However, some definitions computed in this way overlap. For example, a register may be written to by two different instructions within a single BB. To account for cases like these, all definitions are given unique entries in the global definitions list, however only the definition that makes it out of the basic block is retained in the gen set. Thus, the gen set ends up being the set of definitions that are valid at the exit of the basic block.

After all basic blocks have been considered, the GDL contains a list of all operands that are defined in the CT, and all gen sets have been computed. At this time, BB-specific kill sets are computed in the following manner: We iterate through each definition in the gen set of the basic block and compare it to each definition in the GDL. If the gen set definition "kills" the GDL definition (Section 4.6.5), the GDL definition is added to the BB kill set. In this way, we generate kill sets for each basic block in the CT.

Once we have BB-specific gen and kill sets, we apply the standard reaching definitions algorithm [22] to find the reaching definition in-sets (RDins) for all basic blocks.

4.6.5 Criteria for Killing Definitions

At many stages of our slicing algorithm, we need to test for whether an operand affects the value of another. This is similar to the problem of deciding whether one definition kills another. We therefore define a set of criteria we use to decide whether an operand (or definition) affects another.

A definition a is considered to kill a definition b if they are both identical registers or memory references. In addition, a is considered to kill b if b is a memory reference, and a is a register, and b depends on a . This is because the act of writing a causes b to change.

The above criteria work well, except in the case of aliasing. They do not detect the case in which two memory operands have different base registers and operands, but the same effective address. In our work, it is important for the prefetch slice to be as small as possible so it can run ahead of the main thread. For this reason, we choose an aggressive approach and ignore the possibility of aliasing so as to minimize the number of instructions added to the slice. Although there might be cases in which this is not the correct approach, the speculative nature of the assist thread ensures that program correctness is maintained nevertheless. In the worst-case scenario, the assist thread may not prefetch at all, or may prefetch a wrong address and possibly pollute the cache.

As a future optimization, aliasing may be handled. In fact, since ours is a wholly runtime system, we may be able to use runtime program information to resolve aliasing, and may not have to be too conservative in our approach. At present, one of our benchmarks (*mcf*) experiences problems due to uncaught aliasing. In this case, we choose to ignore the basic blocks which cause this and construct our CT and slice without them. We still maintain program correctness, and our prefetch slices still run long enough to perform useful prefetching.

4.6.6 Generating the Slice

We use the results of the global reaching definitions analysis to perform slicing. Our slicing algorithm is based on delinquent load instructions. The final slice is to consist of: the delinquent loads, all instructions that are involved in computing the value of the delinquent load, and all control flow instructions that affect computation of delinquent loads. By extension, instructions that affect control flow by setting flags (e.g., compare instructions, etc.) also need to be added to the slice. Finally, instructions that cause the slice to go offtrace also need to be included. Our slicing algorithm has been designed to account for all the above requirements.

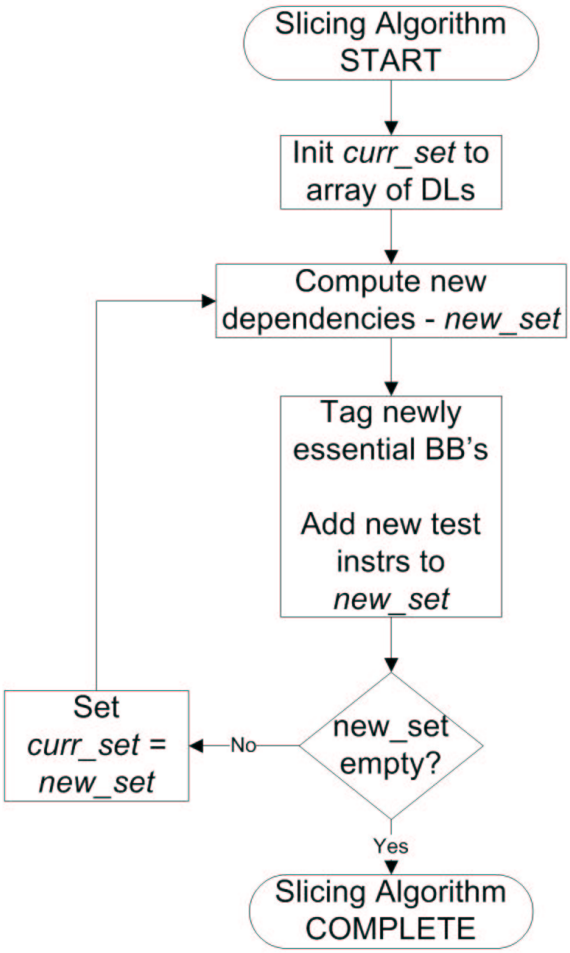


Figure 4.8: Slicer Loop

The slicing algorithm is iterative - each iteration starts with a set of instructions that need to be added to the slice, *curr_set*. Each instruction in this set is added to the slice. If this is the first time an instruction is being processed, instructions *it* depends on are added to *new_set*. If the instruction has been processed before (i.e., the instruction, and by consequence, instructions it depends on, have already been added to the slice), it is ignored. This way, after all instructions in *curr_set* have been processed, *new_set* contains a set of instructions that will be processed in the next iteration of the slicing algorithm.

Each iteration, we process control flow instructions and instructions that affect flags (i.e., instructions that indirectly affect control flow). In this manner, we slice for data flow as well as control flow.

Finally, at the end of the iteration, we set *curr_set* to *new_set* and begin the next iteration. We stop slicing when no new instructions have been added, i.e., when *new_set* is empty. At this point, the prefetch slice has been generated. However, it still needs further processing.

Computing Instruction Dependencies

Computing instruction dependencies refers to the process of finding all instructions which affect the result of an instruction (*targ*) that we are interested in. Each source operand of *targ* is analyzed. Immediate operands are ignored because no instruction can affect them. Once we have decided that a source operand needs to be processed, the reaching definition in-set for *targ* (*instrRDin*) is computed.

We need to find the *instrRDin* because the global, basic-block-specific reaching definitions analysis is insufficient. This is because instructions in the same basic block, but executing before the instruction under consideration, may generate or overwrite (kill) definitions. We use the results of the global analysis and local gen and kill information to generate an *instrRDin*on an as-needed basis for each requested instruction.

A complete *instrRDin*contains the set of all operand definitions that reach *targ*. We

compare each source operand of *targ* to the definitions in *instrRDin*, and use our criteria (Section 4.6.5) to see if any definitions affect the source operands of *targ*. Since each definition is generated by a specific instruction, it is now a simple matter to compute the set of all instructions that affect *targ*. We have thus computed dependencies for the instruction in question.

Tagging Essential Control Flow Instructions

Essential control flow instructions (CFI's) are those which select between paths that could alter delinquent load computations. A basic block X contains essential CFIs if there is any slice instruction I between it and its immediate postdominator. Although X may not compute any values for the slice, it is essential because it chooses between its successors, which could affect whether or not I is executed or not, thus changing the DL address computation. For this reason, the test instruction in BB X (the last instruction in the BB that affects the eflags register) needs to be added to the slice, along with the BB-ending jump instruction(s). We therefore tag essential BBs, and add the test instructions of tagged BBs to the slice (along with dependencies). The jump instructions are handled separately (Section 4.6.8)

For example, in Fig. 4.9a, the BB is essential because one of its successors contains an essential instruction. However, in Fig. 4.9b, the BB is not essential because all its successors upto its postdominator do not contain an essential instruction.

Essential BBs are computed as follows: As instructions are added to the slice, their respective BBs are tagged as essential. We then recursively traverse all paths between the basic block X and its immediate postdominator X . If any path contains an essential BB, then BB X is also tagged as essential. We repeat this tagging process each iteration of the slicer, since we may have added more slice instructions (and thus, more directly essential BBs).

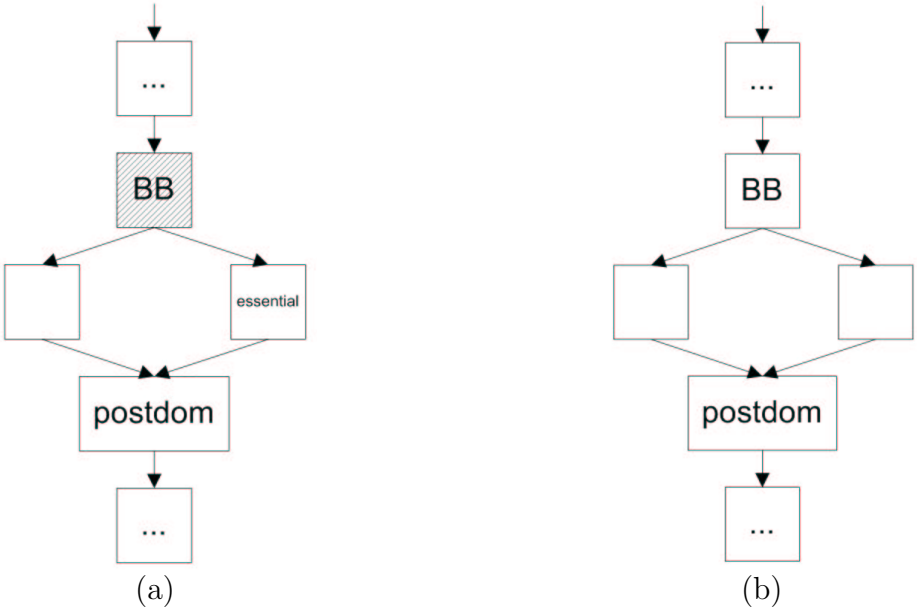


Figure 4.9: An example of essential BBs

4.6.7 Memory Write Redirection

The assist thread is a speculative thread that has a subset of the code in the main thread. In addition, the assist thread is intended to "run ahead" of the main thread. Therefore, it is essential to ensure that the assist thread does not make any modifications to memory, as that may affect correctness of the main program. For this reason, the prefetch slice must not contain any writes to the memory space of the main thread.

We allocate a "scratchspace" which the assist thread can write to. Once the prefetch slice has been generated, we iterate through it and search for any memory writes. Memory writes are redirected to the scratchspace in the following manner: First, a new memory operand with a unique scratchspace address is generated. Then, the original memory write operand is replaced with the new operand. Both original and new operands are saved in a structure that keeps track of memory write redirections.

Since we have redirected memory writes, we also need to patch certain memory reads to read from the scratchspace rather than from main memory. To accomplish this, we again iterate through the slice, searching for any reads from memory that depend on

redirected writes. The respective read operands are then replaced with scratchspace operands.

Sometimes, replacing memory reads and writes may cause correctness issues. For example, when the assist thread runs, a patched read may be executed before its respective patched write. In this case, the redirected location has incorrect data. To prevent this, we prime the scratchspace before invoking the assist thread - we copy data from original memory locations to their equivalent scratchspace locations before executing the prefetch slice.

As a further optimization, and to minimize cases where both main and assist threads access the same memory location, we redirect invariant memory reads in the slice. An invariant memory read is one whose effective address does not change throughout the duration of the slice. This could be a static memory read or a base-indexed read whose base register is not written to anywhere in the slice. We iterate through the slice searching for invariant reads, and replace them with reads from the scratchspace. We keep track of invariant reads that are being replaced with scratchspace reads, so that the scratchspace can be primed before the prefetch slice is executed.

4.6.8 Tracking Jumps

Most jump instructions encode the target address in their opcode. The fragments we use to generate the prefetch slice are part of the connected trace. For this reason, jump instructions in the prefetch slice still target PCs in the trace cache. We need to retarget any jump instruction in the slice so that it points to the appropriate instruction in the slice itself.

We retarget jumps in two stages. The first stage occurs after the slice has been generated. We iterate through the list of basic blocks, and for each essential basic block, we add its terminal jump trampoline (which is a set of three jump instructions) to the final slice. In the process of adding the jumps, we encode the "note" (which is a user-

customizable field attached to each dynamorio *Instr* data structure) to reflect the fact that the instruction is a jump and also encode the serial number of the target instruction in the note field. This note will be used in a subsequent step. We handle offtrace jumps by encoding the note with a special number that denotes a jump to exit. The final encoding of the note field of any instruction contains:

- The instruction's own unique serial number.
- Information as to whether the instruction is a jump or not.
- For jumps to other instructions in the slice, the serial number of the target instruction.
- For offtrace jumps, a unique number denoting a jump to exit.

In addition to encoding jumps, we also try to perform a simple optimization here. As we have seen in section 2.1.2, an artifact of Strata is that it converts all conditional jumps to three-jump trampolines. In our slice, we do not need the flexibility of a trampoline. Therefore, in simple cases where a basic block successor is the next basic block in the list, we try to eliminate the trampoline and just leave a single conditional jump. At this time, we do not perform non-trivial optimizations which may involve changing the sign of conditional jumps. This would reduce slice size further.

The second stage of jump retargeting occurs just before the prefetch slice is encoded in memory (Section 5.3). We iterate through the entire slice and decode the note field of each instruction. For all jump instructions, we use a DynamoRio function to set the target to the appropriate target instruction. Once we have set the targets, DynamoRio automatically sets jump opcodes when it encodes the slice into machine code.

4.7 Prefetch Slice Generation Summary

In this section, we have described the process of generating a connected trace. We have introduced the criteria for deciding what constitutes a CT and described how a CT can be expanded. We have also enumerated the steps involved in generating a prefetch slice using a CT. We have introduced some issues that arise during slice generation and presented our solutions. In the next chapter, we will show how a prefetch slice is invoked and executed.

Chapter 5

Executing and Coordinating the Assist Thread

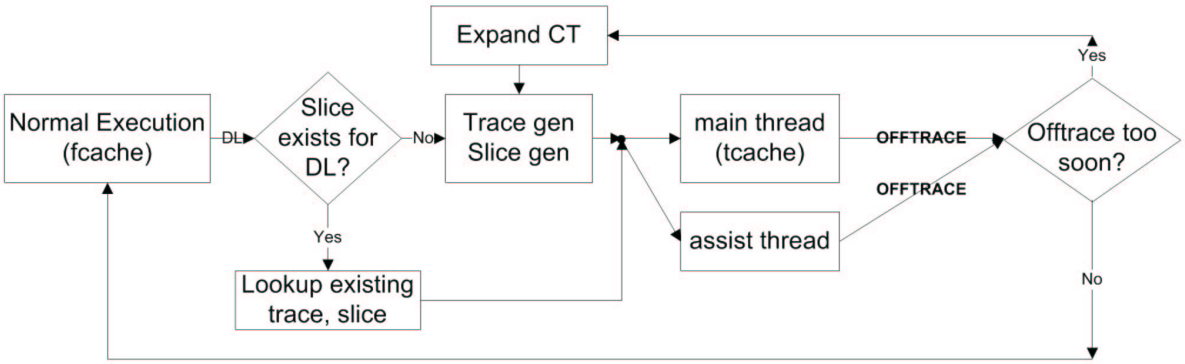


Figure 5.1: TOPP working flowchart

5.1 Overview of Assist Thread Coordination

In the previous two chapters, we have described our techniques to generate a prefetch slice for a particular delinquent load. In this chapter, we describe ancillary procedures that are not directly involved in generating the prefetch slice, but are necessary to properly execute and coordinate the assist thread.

We discuss synchronization of the assist thread and the main thread and identify relevant issues. We then discuss how an assist thread is executed using a generated prefetch slice. We present optimizations such as reusing previously generated prefetch slices (Fig. 5.1).

5.2 Synchronization of the Assist Thread

Synchronization of the assist thread is a very important issue. The utility of the assist thread is in its function of prefetching. Therefore, it is necessary to ensure that the assist thread always runs ahead of the main thread. We do this by slicing, to pare the assist thread down to the minimum number of necessary instructions. However, since the assist thread is expected to run faster, it needs to be kept only a few iterations ahead of the main thread.

If the assist thread is too far ahead, prefetched data could begin to evict cache blocks that are still needed by the main thread [13]. For this reason, the prefetch slice needs to constantly check where its execution is, in relation to the main thread, and accordingly wait when it is too far ahead.

Our current implementation of TOPP does not handle the case where the assist thread is too far behind. Such a case is, in fact, quite unexpected, considering that the prefetch slice is usually much smaller than the equivalent code in CT.

We implement synchronization the following way. While generating the connected trace, when we are generating the primary DL, we add an instruction to increment a counter, the main thread iteration counter. This counter counts the number of times the primary DL instruction is executed. Similarly, in the the prefetch slice, we insert code just after the instruction that corresponds to the primary DL. to increment an assist counter that corresponds to the number of times the prefetch slice has executed the primary DL. Comparing the values of the main and assist counters helps maintain

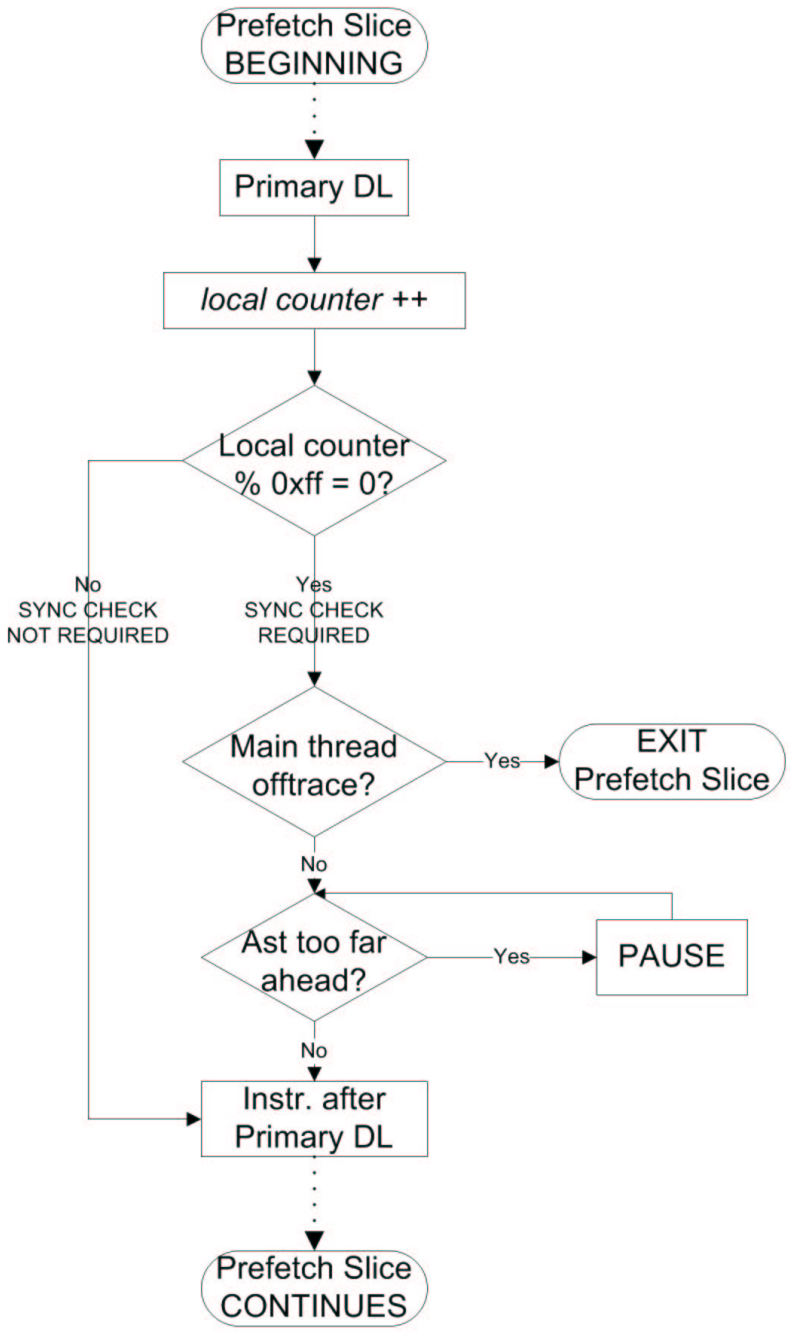


Figure 5.2: Assist Thread Synchronization Flowchart

synchronization.

However, checking for synchronization has its own overhead, especially as shared variables (the counters) are being constantly changed and read by different threads. To minimize overhead, we implement chunk prefetching. We only compare the main and assist counters when the assist counter's last byte is zero. In other words, we only perform synchronization checks every 256 increments of the assist counter. We chose the number on the basis of experimental timing measurements, and also because it provides a tolerable tradeoff between accuracy and overhead. It must be kept in mind that chunk prefetching does not affect correctness. i.e., the prefetch slice will go offtrace and exit when it is supposed to, even if that happens in the middle of a chunk. Chunk prefetching only affects the synchronization check itself.

Therefore, the prefetch slice checks synchronization as follows:

1. Execute primary DL prefetch instruction
2. Increment assist counter
3. If last 8 bits of assist counter are all zero, synchronization check is needed
4. Synchronization check: Compare local and global counter. If too far ahead, pause and recheck. If not, continue executing slice code.

There are two synchronization parameters that are variable. They are:

Slack: This is the number of iterations that the assist thread is allowed to run ahead of the main thread. If this value is too low, the assist thread will constantly start-stop, each time checking the main thread counter. This is not desirable as it causes excessive synchronization overhead. If the slack is too much, the assist thread could fill up lower levels of cache and evict blocks that the main thread hasn't read yet. This is obviously counterproductive and must be avoided.

Pause Duration and Type: The assist thread needs to be paused when it is too far ahead. There are two ways to do this - using the *usleep()* call or using a number of *pause* instructions. We found that the *usleep()* call has a minimum sleep period of one OS "tick". However, its advantage is that it it completely deactivates the logical CPU of the assist thread, thus causing no overhead on the logical CPU of the main thread. On our test system, one OS tick equates to a minimum of 20 milliseconds. Because we require pause times of a few hundred cycles, we are unable to make use of usleep(). The second way to pause a thread is to us a *pause* instruction [23]. We use a number of consecutive pause instructions to pause the assist thread. Unfortunately, the logical CPU executing the pause instruction is constantly active, and has some impact on the performance of the main thread.

In Section 6.5.3, we present experimental results, showing the effect of parameters such as slack and pause duration on TOPP's performance. We choose our parameters based on these results.

5.3 Completing the Prefetch Slice

As discussed in Chapter 4, a prefetch slice consists of a set of instructions that are necessary to compute and prefetch delinquent loads. However, in addition to simply generating the prefetch slice, additional instructions need to be added to get a self-contained, fully functioning prefetch slice. The code added is considered "housekeeping" code as it is not directly involved in prefetching, yet it is necessary to ensure that the prefetch slice can actually execute properly. The code described in this section is executed when the prefetch slice is invoked. The following steps are involved in adding housekeeping code:

1. The slice is analyzed to find two registers that are not accessed in any part of the slice. These registers will be used to keep track of the assist counter and synchronization. Under the current implementation, if no "safe" registers are found, slice

generation is aborted and normal fragment cache execution resumes. Fortunately, our test benchmarks do not encounter this issue due to the fact that the stack pointer and frame pointer are unchanged except across function calls, which we do not encounter in slices generated from our test benchmarks. A slightly slower but safer alternative would be to use memory locations or register spilling instead of looking for safe registers.

2. We insert instructions to save registers and flags, and copy the appropriate register and flag values from the main thread. This ensures that the prefetch slice starts with the same context that the main thread had at the time of DL identification.

3. We insert code to prime the scratchspace (See Section 4.6.7).

4. After the code to prime the scratchspace, we insert an indirect jump instruction. The purpose of this jump instruction is to start the prefetch slice at a point for which its initial context is valid. Section 5.4.1 explains this concept in detail.

5. After the indirect jump comes the actual prefetching slice code. We insert synchronization code, including the pause loop, immediately after the primary DL instruction in the slice. Along with the synchronization code, we insert an offtrace check - to exit the assist thread as soon as the main thread goes offtrace. This is not a common occurrence since the prefetch slice usually runs ahead of the main thread; it usually goes offtrace and exits before the main thread.

6. We append exit instructions at the end of the prefetch slice to restore the original context of the assist thread, and a return instruction. In this way, the entire prefetching slice, including housekeeping code, can be invoked using a function call, using a function pointer.

7. As an aside, since we have information about the exit basic block, we perform jump retargeting (Section 4.6.8) at this point.

```
while(1)
{
    pthread_cond_wait();
    slice_fptr(); //Prefetch slice function
    //End of prefetch slice, loop back
}
```

Figure 5.3: Assist thread invoke loop, runs on the second logical CPU

The above steps complete the prefetch slice, and make it a self-contained entity that coordinates and prefetches without further intervention. However, it is still in the form of a DynamoRio instruction list. We use DynamoRio's encode facility to encode the list into machine code in memory. The starting address of the memory block is thus the starting address of the prefetching slice. To invoke the prefetch slice, a function pointer is set to the starting address and called.

5.4 Invoking Prefetch Slices

In our implementation of prefetch slice generation, the CT and slice are generated while the main thread is interrupted. The prefetch slice is intended to be executed on the same physical CPU, by a different thread.

When the program is started, we create and start the assist thread using pthreads. However, we keep the assist thread waiting for a conditional signal. After the prefetch slice is completed, the main thread sets a function pointer to the starting address of the newly generated prefetch slice. It then signals the assist thread. The assist thread immediately executes a function call to the function pointer, which, in effect, starts executing the prefetch slice on the same logical CPU as the assist thread. The main thread proceeds to execute fragments within the CT.

When the prefetch slice completes by going offtrace, the assist thread resumes waiting for another signal. When the main thread goes offtrace, TOPP exits prefetching mode and reenters strata mode (i.e., fragment cache fragments are now executed). This approach also lets us repeat the process as many times as desired, letting us reuse prefetch slices. Fig. 5.3 and Fig. 5.1 describe this process.

5.4.1 Prefetch Slice Start Points

As discussed in Section 3.3, any DL may initiate prefetching. Since context (registers and flags) are only valid at a particular intersection, we need a mapping that describes where to jump to in a slice for a given DL. We implement this by generating a mapping between tcache addresses for the CT and prefetch slice addresses at the same time the prefetch slice is generated. Each time a prefetch slice is invoked, the mapping is used to find the appropriate start point in the slice. We load a memory location with the appropriate start point address. As discussed in section 5.3, an indirect jump in the beginning of the completed prefetch slice makes the slice execution start from the correct point.

5.5 Reusing Prefetch Slices

In most real-world programs, there isn't just one loop - there are nested loops, functions called repeatedly, etc. For this reason, connected traces are usually invoked repeatedly, and consequently, the prefetch slice needs to be started and stopped repeatedly. In addition, programs have different CTs which may execute in turn, also repeatedly. For the above reasons, it has been necessary to implement support for reusing prefetch slices and connected traces.

We use a hash table to keep track of CT's and slices for reusing. Each time a slice is generated, an entry is made in the hash table for the primary DL. In addition, entries are also created for secondary DLs. Since there is some shared and common information

between primary and secondary DLs, we use a two-structure approach:

dl_struct: The first data structure is the hash table entry itself. There is one of these for every DL, primary or secondary. This contains just the fragment cache DL address, the location of the DL in the CT (the trace cache DL address, in other words), and a pointer to a *trace_struct*.

trace_struct: The second data structure is specific to each trace. All *dl_structs* belonging to a trace have a link to the appropriate *trace_struct*. The *trace_struct* contains the following information:

- The starting address of the prefetch slice.
- A list of all the trace cache fragments that make up the connected trace
- A list of all the primary and secondary DLs in the CT.
- A list of all possible startpoints and their corresponding startpoint addresses in the slice.

Using the above hash table structure, we are able to reuse CTs and slices. When we identify a DL, we do a hash table lookup based on its fragment cache PC. If we find it, we load the corresponding prefetch slice into the function pointer (Section 5.4). We then use the startpoint array for the CT to calculate the appropriate assist startpoint and set the indirect jump variable accordingly. We then reset the main iteration counter and start off the slice and start executing fragments from the CT.

Using the above method, we can repeatedly reuse slices and traces without having to regenerate them. We can also expand slices and CTs (Section 4.4) after a number of reuses, as our hash table structures store all the information needed to expand a CT at any time.

5.5.1 DL Patching

As we have complex control flow in real-world programs, we need a method to identify if we are back on a DL-containing trace. Once we have generated a CT and slice, we need a way to ensure that we re-identify the DL and reuse the slice at the correct time. We do this as follows:

After a CT and slice are successfully generated, we patch the fragment cache DL instruction by code that calls the Strata fragment builder. Each time execution reaches that point, we immediately enter prefetching mode and reuse the trace and slice. This provides perfect coverage of the delinquent load trace. However, there is one shortcoming of this approach in that we do not detect delinquent loads on the CT any more. Thus, we are not able to detect any new loads that may take longer to be identified as delinquent and continue prefetching for only the original set of DLs. Because of this, we may miss some DLs for the entire execution of the program.

5.6 Summary of Assist Thread Coordination

In this chapter, we have described how the pieces of our architecture fit together. We have introduced the concept of assist thread synchronization and discussed our approach to it. We have presented how we complete a prefetch slice and prepare it for execution. We have also discussed the issues concerning invoking a prefetch slice with the proper context and presented our solutions to them. Finally, we have discussed issues concerning real-world benchmarks and our approaches to efficiently dealing with multiple traces and slices.

Chapter 6

Methodology and Results

6.1 Overview of Methodology and Results

In this chapter, we describe our experimentation and evaluation procedures on TOPP. We first introduce our test system and its characteristics. We then touch upon the parameters we use to evaluate TOPP. We describe the different benchmarks we use to analyze the performance of TOPP. We present the performance results of TOPP, and discuss how it is influenced by external overheads and internal parameters. We conclude with suggestions for future optimizations to TOPP.

6.2 Test System

Our test system is a Dell PowerEdge 6600 server. It has four Intel Pentium IV Xeon CPUs running at 1.6 Ghz, each with hyperthreading enabled, and 2 GB of RAM. We use RedHat 7.3 with a Linux kernel version of 2.4.18-3. We have modified the kernel to accomodate our purposes. We have implemented thread binding - where a thread or process is bound to a single CPU. This is necessary because performance monitoring and prefetching both require that threads involved do not migrate. We have also added system calls to the kernel to allow programs to access performance monitoring registers

and counters. Finally, we have patched the kernel to enable PEBS interrupts [24].

6.3 Evaluation Parameters and Methodology

We evaluate TOPP using a set of benchmarks. We focus on wall clock time and cache miss count as the two main parameters used to compare TOPP-enabled benchmarks to the equivalent native versions.

We use the *time* application to measure wall clock time. *time* also keeps track of the total time used by all individual threads of a process - we use this to compute the percentage of time the assist thread was active and to find the percentage of program execution that was covered in prefetching mode.

To measure cache misses, we use another set of performance monitoring registers. At the time TOPP is initialized, we activate a counter to count the number of L2 cache misses as seen by the main thread. We read the counter just before the program exits. We take care to avoid conflicts between this performance monitoring and TOPP's inbuilt dynamic profiling.

6.4 Benchmarks

We use a set of benchmarks to evaluate TOPP. Since the purpose of TOPP is to prefetch into the cache, we focus on benchmarks that are expected to be memory-intensive. Among these, we evaluate TOPP on both pointer-chasing benchmarks (e.g., list and tree access) and array-based applications. We also evaluate TOPP on a real-world benchmark. The following are the benchmarks we use:

list

List is a simple list-based benchmark that focuses on list access. It consists of an allocation phase, where a large circular linked list is allocated and linked. Then, the access

phase is started - the list is iterated through. At each iteration, a small amount of work (floating point arithmetic) is done. Since the list is circular, it can be iterated through as many times as required so that we have the benchmark run for a significant enough amount of time. TOPP is invoked at the beginning of the access phase. It must be noted that we do not artificially randomize the list - in practice, we find that successive list elements are more or less contiguous in memory. Randomizing the list would probably inflate performance measurements in favour of TOPP, however it would not reflect real-world applications.

array

Array is a simple array-based benchmark. It is similar to the *list* benchmark in having an allocation and access phase. It also has floating point work inbetween memory accesses.

BST

BST is a binary search tree benchmark. In the allocation phase, we allocate and initialize a large binary search tree. In the access phase, we perform many repeated random searches for data from the tree, on which we perform some floating point arithmetic to simulate work.

mcf

Mcf is a real-world benchmark that is part of the Spec CPU 2000 integer benchmark suite. We chose *mcf* because of its relatively high proportion of memory access operations and relatively high cache miss rate, as compared with the remainder of the Spec suite. However, since it is an integer benchmark, it is expected to be suboptimal in terms of taking advantage of hyperthreading. In Intel's implementation of hyperthreading, functional units such as adders, floating point units are shared between the two logical CPUs. For this reason, hyperthreading works well if the two threads perform different

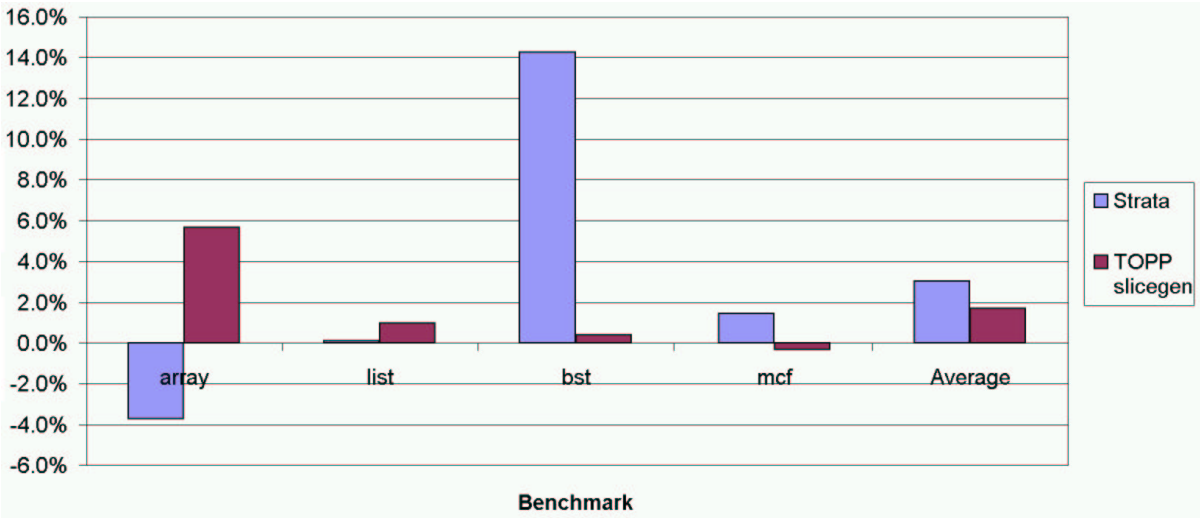


Figure 6.1: Strata and Slice Generation Overheads

types of computations - e.g.,one thread performing mostly integer operations, and the other, floating point operations.

6.5 Results

In this section, we discuss the results obtained from running the benchmarks described in Section 6.4. We analyze our results using the parameters introduced in Section 6.3.

6.5.1 TOPP Overheads

Since TOPP involves a number of runtime activities, there are significant overheads incurred in to process of generating and executing a prefetching slice. These are:

Overhead due to Strata

This occurs because of Strata's interpretation, fragment generation and also because Strata inserts extra trampoline jumps. As we see in Fig. 6.1, Strata's overhead varies between 0.1% for *list* to over 14% for *bst*. This is because of the overhead of instruction

decoding and fragment generation. In fact, the average Strata overhead for the Spec CPU 2000 benchmark suite is about 35% [3].

TOPP overhead

This includes the overhead of performance monitoring and PEBS interrupts, and also that of generating connected traces and prefetch slices. This is normally quite low, since performance monitoring is quite lightweight, and CT and slice generation happen only a few times over the entire execution of a program. As seen in Fig. 6.1, TOPP overhead is usually negligible, although it is as high as 5.7% for *array*. The slowdown can be attributed to performance monitoring overhead, which can be quite significant, the longer it takes to identify a DL. In fact, *array* spends about 14% of its time with performance monitoring activated, which is very high when compared to *list* , which spends less than 1% of its total execution time monitoring performance.

Hyperthreading overhead

Threads that run on two logical contexts of the same hyperthreaded CPU can slow down significantly as a consequence of resource sharing between the two logical CPUs. In Intel's design of hyperthreading, some CPU resources are statically partitioned as soon as the second logical thread context becomes active [2]. For this reason, a logical thread executing even only no-ops can negatively impact the first logical thread. Along with static resource partitioning, instruction mix also determines slowdown due to hyperthreading. For example, if the two logical threads are running identical code, there is a greater potential for slowdown as functional units are limited in number. On the other hand, hyperthreading shines when the two threads are running code that use different functional units (e.g., integer vs. floating point operations). For this reason, instruction mix is an important factor in determining hyperthreading overhead.

Although it is not possible to directly measure the overhead of the assist thread (due

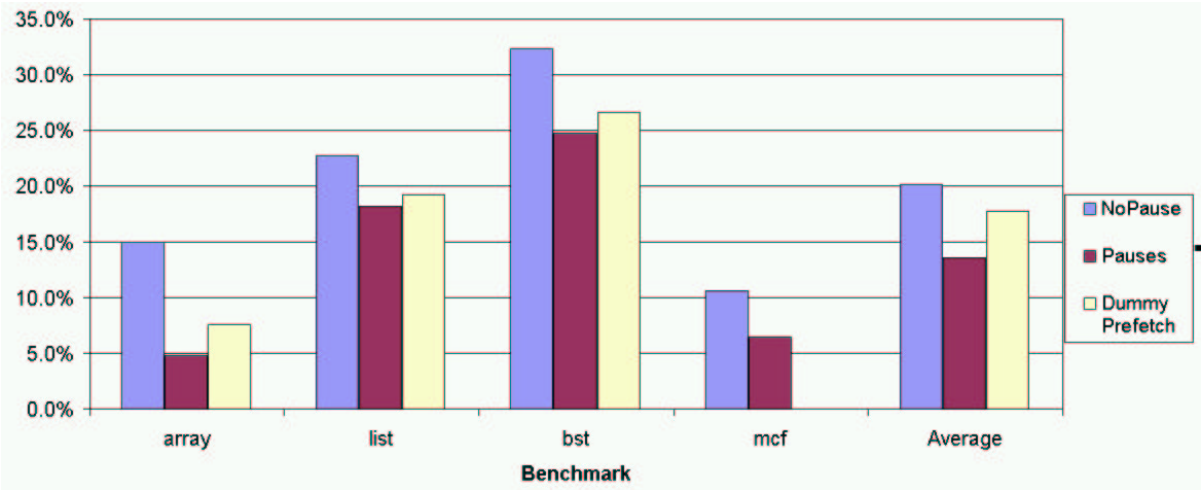


Figure 6.2: Hyperthreading Overheads

to the fact that the assist thread is simultaneously speeding up the main thread by prefetching), we obtain an estimate of hyperthreading overhead by simulating an assist thread. We have a small program whose only function is to loop, and in each iteration of the loop, perform some integer math and access a single memory location, a mix of instructions which is quite similar to a simple assist thread. We measure the slowdown caused by running both the assist thread simulator and the our benchmarks on the same physical CPU but on different logical processors. We use two versions of this program to simulate extreme cases hyperthreading overhead. The "NoPause" version simulates a constantly executing assist thread with no pauses. The "Pause" version contains *pause* instructions embedded in the main loop that effectively reduce the load of the program on its logical CPU. This version simulates a scenario where the assist thread is pausing roughly half the time and executing for the other half. We do not realistically expect hyperthreading overheads to be either greater than the "NoPause" case or lesser than the "Pause" case. Figure 6.2 shows the hyperthreading overheads for each benchmark measured in this manner. These overheads are as compared with native versions of the benchmarks.

For the microbenchmarks *array* , *list* and *BST* , we use an alternative method of

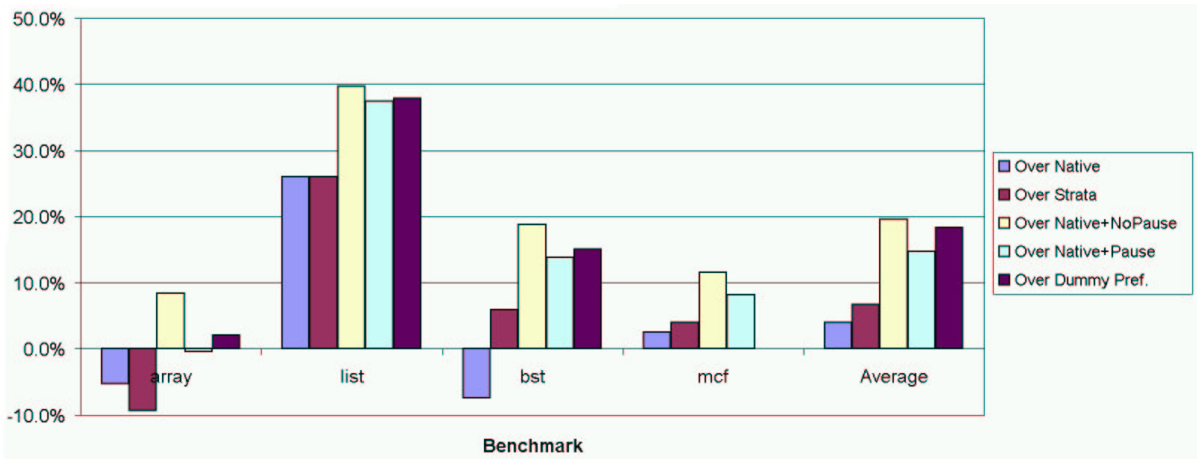


Figure 6.3: Speedup achieved due to TOPP

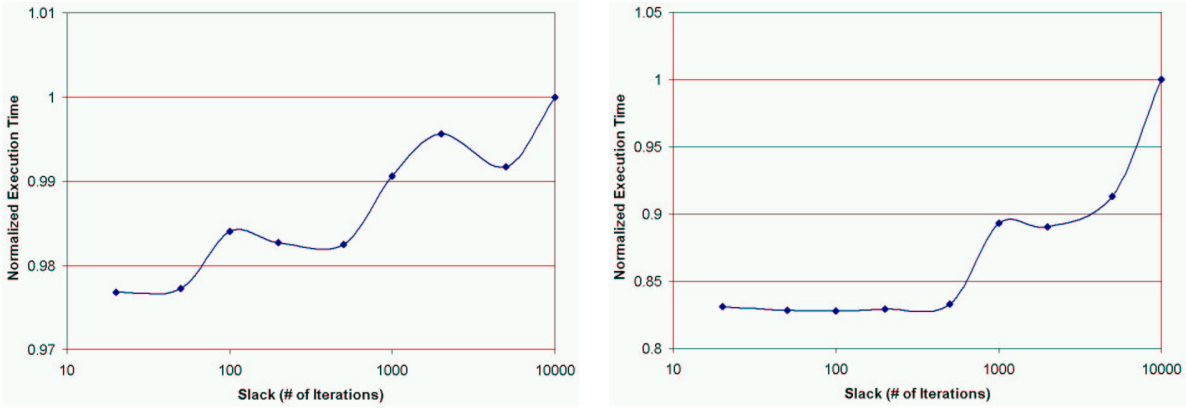
measuring combined overheads - we generate the prefetch slice as usual, however, in the prefetch slice, we replace the instruction that corresponds to the delinquent load (i.e., the instruction that actually does the prefetching) with a *nop*. Thus, we have a very accurate way of measuing the overhead of generating and executing the prefetch slice while isolating its beneficial effects. However, this is only possible for the microbenchmarks as they do not have multiple delinquent loads. As seen in Figure 6.2, hyperthreading overhead as measured with the dummy prefetch method is slightly greater than that measured using the "Pause" method above for all the microbenchmarks.

As seen above, TOPP has to contend with hyperthreading overheads ranging between 5% and 25%.

6.5.2 Performance Improvement

We present the performance impact of TOPP in Fig. 6.3. As seen in the figure, TOPP results in a speedup over native code in two of four cases, and a speedup over Strata in three out of four cases.

If we consider the overhead due to Strata and hyperthreading, either of which we have no control over, total overhead ranges between 7% for *array* to over 27% for *bst* (as



a) *mcf* b) *list*

Figure 6.4: Influence of Slack on Speedup due to TOPP

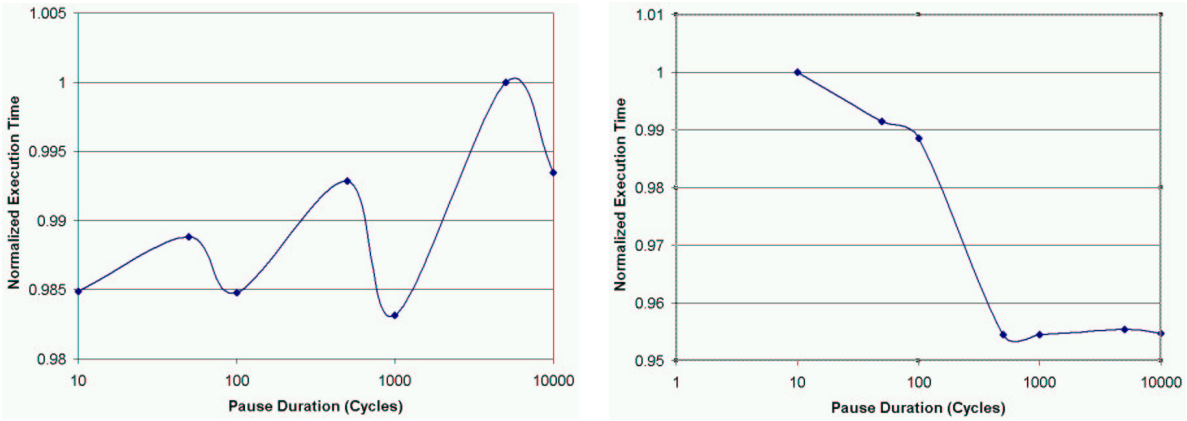
calculated from the dummy prefetch method in Fig. 6.2). Thus, as can be seen in the results, TOPP gains back most of this overhead, for all benchmarks. In the next section, we look at the influence of TOPP parameters on performance.

6.5.3 TOPP Parameters

As discussed in earlier sections, a number of variable parameters may influence TOPP's performance. In this section, we attempt to determine which parameters influence performance, and we determine the level of influence of each variable parameter. We consider the following parameters:

Slack

As seen in section 5.2, slack refers to the number of iterations that the assist thread is allowed to run ahead of the main thread. As can be seen from Figure 6.4, too much slack can be detrimental to performance. This could be because the assist thread runs so far ahead of the main thread that it evicts cache lines that have not yet been accessed by the main thread. The effect is more apparent on a benchmark with long loops, such as *list*, rather than *mcf*, which has many short CT invocations, meaning that the assist thread does not have the opportunity to get too far ahead before an offtrace condition occurs.



a) *mcf* b) *list*

Figure 6.5: Effect of pause duration on performance

We choose a slack of 50 iterations based on the results in Fig. 6.4.

Pause Duration

From Section 5.2, pause duration refers to the amount of time an assist thread waits if it is too far ahead. We implement the pause using multiple instances of the IA-32 instruction of the same name. We vary the duration by varying the number of *pause* instructions the assist thread waits for. From Figure 6.5, we see the effect of pause duration is different for different benchmarks. We choose a pause duration of 1000 cycles (i.e., we insert 1000 *pause* instructions at the appropriate place in the slice). We choose this number because it gives us the best performance numbers for both *list* and *mcf*.

Secondary DL Threshold

A secondary DL is one that has not been identified as delinquent, but has a significant kernel-level count - denoting that it too causes cache misses (Section 4.6.1). The Secondary DL Threshold refers to the number of samples at which a DL is considered secondary. It is expressed as a fraction of the primary DL threshold. Upon varying seocndary DL threshold, we notice a decrease in performance for lower thresholds (Fig. 6.6). This is because, at low thresholds, many DLs are declared as secondary, causing bloat in

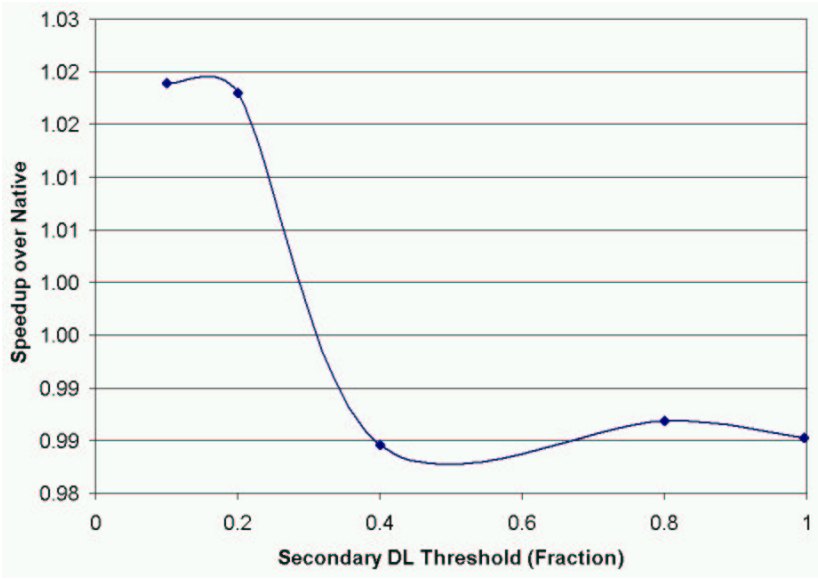


Figure 6.6: Effect of Secondary DL Threshold on Speedup on *mcf*

the prefetch slice, which is then not able to always run ahead of the main thread. This shows that adding too many DLs to the prefetch slice can be counterproductive.

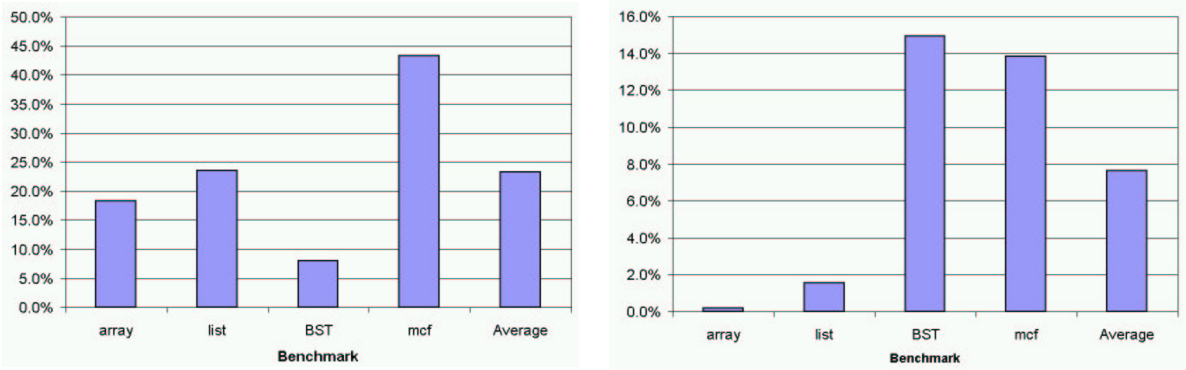
6.5.4 Cache Performance

Figure 6.7(a) presents load/store instructions as a percentage of all instructions executed by a benchmark. This tells us which benchmarks have memory access as a significant component of their execution, and which benchmarks are more CPU-intensive.

Figure 6.7(b) presents the cache miss rate for each benchmark, as measured without TOPP. This gives us a better idea as to why some benchmarks, such as *list* show significant speedups with TOPP whereas others, such as *array*, do not.

TOPP greatly improves cache performance for each of our chosen benchmarks. Cache misses are reduced by a minimum of 36% (*mcf*) to over 99% (*list*). Figure 6.8 presents cache miss reduction achieved by TOPP for each of our benchmarks.

Thus, we see that TOPP has the greatest impact on programs like *list* which have significant time spent in memory access and high cache miss rates. TOPP is usually



a) Load-Store Ratio b) Cache Miss Rate

Figure 6.7: Cache Performance of Native Benchmarks

able to significantly mitigate cache miss rates on such programs, thus achieving speedup. On programs such as *array* , which do not have such high cache miss rates, TOPP still drastically reduces cache misses, however the reduction in cache misses does not necessarily translate to a significant speedup.

6.6 Comparison with Hardware Prefetcher

The Intel Pentium IV Xeon CPUs of our test system are equipped with a built-in automatic hardware prefetcher. This is a stride prefetcher, and is expected to excel in memory-intensive programs with predictable memory access patterns. In fig. 6.9, we compare the performance of the hardware prefetcher with that of TOPP.

As seen in the figure, the hardware prefetcher achieves significant speedups on the benchmarks *array*, *list* and *mcf*, which have regular memory access patterns. Although *list* and *mcf* have list accesses, since the lists are not randomized, the hardware prefetcher is able to compute strides and prefetch for them. On the other hand, for the tree benchmark *bst*, the hardware prefetcher has no beneficial effect as the access patterns of a tree do not form regular strides. TOPP is able to speed up this benchmark, since it relies on precomputation to calculate data addresses.

Running the hardware prefetcher along with TOPP produces mixed results. Although

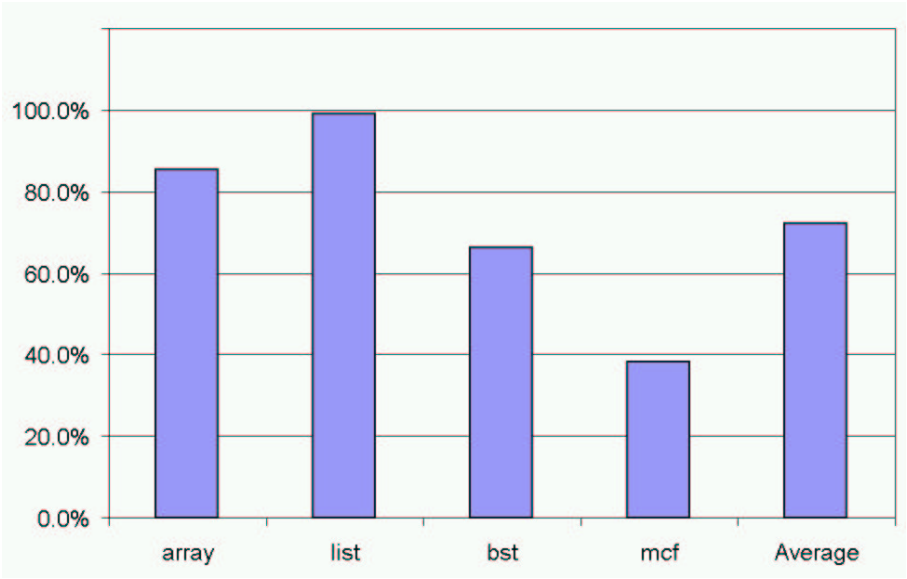


Figure 6.8: Reduction in cache misses due to TOPP

speedups in most benchmarks are better than with TOPP alone, hyperthreading overhead does cause some slowdown for TOPP, when compared with the hardware prefetcher alone. This is also seen in fig. 6.9.

6.7 Future Optimizations

Although TOPP in its present form cannot completely overcome the overheads of underlying tools and hyperthreading, some options are open. First, an SDT with a lower overhead, such as dynamorio, could be used, or an approach that did not need a whole SDT could be attempted. Second, shared cache chip multiprocessors have appeared on the market, especially for laptops. The hyperthreading overhead would be non-existent for these, while the cache prefetching benefits would still be available. TOPP would probably be ideal for such processors. Third, explicit hardware support for assist thread synchronization would go a long way towards mitigating TOPP's inherent overheads.

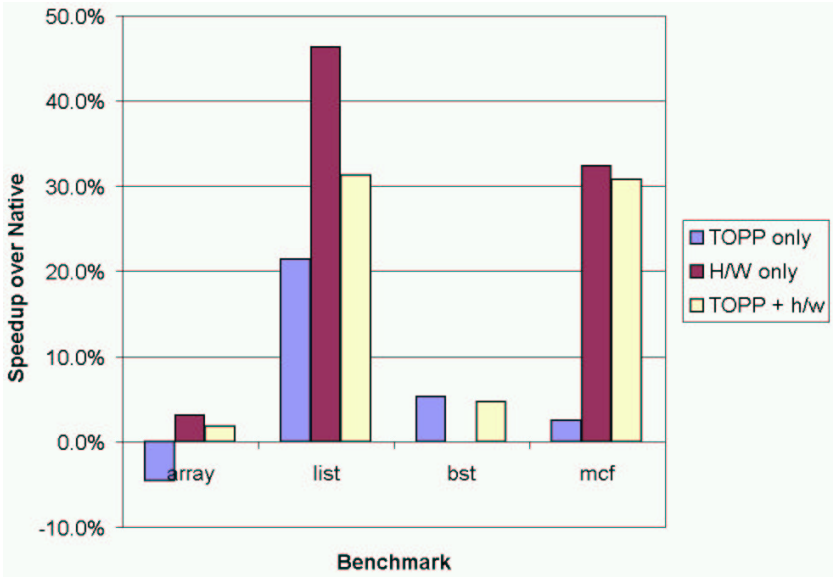


Figure 6.9: Speedup due to hardware prefetcher

6.8 Summary

In this section, we have introduced the benchmarks and performance metrics that were used to evaluate TOPP. We then presented TOPP's impact on the chosen benchmark with respect to two significant performance metrics - wall clock time and cache misses. We found that TOPP has significant unavoidable overheads due to Strata and hyper-threading. Upon analysis of the performance of TOPP-enabled benchmarks, we found that in some TOPP achieves speedups of upto 26% over and above its overheads, and that it slows down some benchmarks by a maximum of 7.5%. We have also demonstrated that TOPP drastically improves cache performance of all benchmarks, reducing cache misses by amounts between 35% and 99%. We have presented an analysis of benchmark characteristics and how they influence TOPP's ability to speed them up. Finally, we have discussed possible improvements to TOPP.

Chapter 7

Conclusion

In this thesis, we have presented TOPP, a novel trace-based system for optimization of programs via prefetching. We have shown the motivation behind TOPP, and discussed what makes it unique compared to existing research. We have shown how we have made TOPP to be a completely runtime system. We have discussed the components of TOPP, and how they interact - we also demonstrate how we use existing SDTs such as Strata to simplify TOPPs tasks. We demonstrate the utility of TOPP on a set of benchmarks, achieving speedups of upto 26%, and reducing cache misses by at least 35%. We now present some future directions for TOPP.

7.1 Future Work

There are two main directions future work on TOPP could take. The first is optimization - i.e. improving the performance of TOPP. At this time, the two greatest overheads on TOPP are caused by Strata and hyperthreading. Using a lightweight SDT with less overhead would alleviate this somewhat. Another option is to avoid using an SDT for the bulk of program execution. In such an approach, we might only activate necessary functionality such as fragment building on an as needed basis, rather than from the beginning of program execution, as we do now. To further study the effects of hyperthreading

overhead, we might try to run TOPP on other SMT CPUs, such as the UltraSparc IV. In addition, TOPP is expected to do well on shared cache CMPs, desktop versions of which are scheduled for commercial distribution very soon.

The second direction for TOPP is extending its capabilities to handle more varied code. For example, extending TOPP to prefetch for recursive loops would open up many more programs for optimization. Adding the capability to detect memory aliasing at runtime would also allow TOPP to work better and with more real-world applications.

Bibliography

[1] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y. Lee, D. Lavery, and J.Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *28th International Symposium on Computer Architecture*, July 2001.

[2] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, J. Alan Miller, and Michael Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 06(01):4–16, February 2002.

[3] K.Scott, N.Kumar, S.Velusamy, B.Childers, J.W.Davidson, and M.L.Soffa. Retargetable and Reconfigurable Software Dynamic Translation. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, pages 36–47, March 2003.

[4] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *1st International Symposium on Code Generation and Optimization*, March 2003.

[5] Intel Corp., Santa Clara, CA. *IA-32 Intel Architecture Software Developer's Manual Volume 3*, 2002.

[6] Mark Smotherman. History of multithreading, 2005.

[7] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, June 1995.

[8] C. K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *28th International Symposium on Computer Architecture*, July 2001.

[9] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *28th International Symposium on Computer Architecture*, July 2001.

[10] A. Roth and G. Sohi. Speculative data-driven multithreading. In *7th IEEE High-Performance Computer Architecture*, January 2001.

[11] Steve S.W. Liao, Perry H. Wang, Hong Wang, Gerolf Hoflehner, Daniel Lavery, and John P. Shen. Post-pass binary adaptation for software-based speculative precomputation. In *ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, June 2002.

[12] Dongkeun Kim and Donald Yeung. An infrastructure for adaptive dynamic optimization, 2003.

[13] Dongkeun Kim, Steve Shih wei Liao, Perry H. Wang, Juan del Cuvillo, Xinmin Tian, Xiang Zou, Hong Wang, Donald Yeung, Milind Girkar, and John P. Shen. Physical Experimentation with Prefetching Helper Threads on Intel's Hyperthreaded Processors. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 27–38, March 2004.

[14] A. Moshovos, D. N. Pnevmatikatos, and A. Baniasadi. Slice-processors: An implementation of operation-based prediction. In *15th International Conference on Supercomputing*, June 2001.

[15] J. Collins, D. Tullsen, H.Wang, and J. Shen. Dynamic speculative precomputation. In *34th Annual ACM/IEEE International Symposium on Microarchitecture*, December 2001.

[16] James E. Smith. Decoupled access/execute computer architectures. In *9th Annual Symposium on Computer Architecture*, 1982.

[17] Yuan Chou and John Paul Shen. Instruction path coprocessors. In *The 27th Annual International Symposium on Computer architecture*, 2000.

[18] Won W. Ro and Jean-Luc Gaudiot. Compiler support for dynamic speculative pre-execution. In *Seventh Workshop on Interaction between Compilers and Computer Architectures*, 2003.

[19] Jiwei Lu, Howard Chen, Rao Fu, Wei-Chung Hsu, Bobbie Othmer, Pen-Chung Yew, and Dong-Yuan Chen. The performance of runtime data cache prefetching in a dynamic optimization system. In *36th International Symposium on Microarchitecture*, December 2003.

[20] Jeffery A. Brown, Hong Wang, George Chrysos, Perry H. Wang, and John P. Shen. Speculative precomputation on chip multiprocessors. In *6th Workshop on Multithreaded Execution, Architecture, and Compilation*, November 2001.

[21] Jiwei Lu, Abhinav Das, Wei-Chung Hsu Khoa Nguyen, and Santosh G. Abraham. Dynamic helper threaded prefetching on the sun ultrasparc cmp processor. In *38th Annual IEEE/ACM International Symposium on Microarchitecture*, November 2005.

[22] Steven S. Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[23] Intel Corp., Santa Clara, CA. *IA-32 Intel Architecture Software Developer's Manual Volume 2*, 2002.

[24] Brinkley Sprunt. Brink and abyss performance counter tools for linux, 2006.