The Dissertation Committee for Kartik Kandadai Agaram
certifies that this is the approved version of the following dissertation:

Prefetch Mechanisms by Application Memory Access Pattern

Committee:

Stephen W. Keckler, Supervisor

Kathryn McKinley

Calvin Lin

Doug Burger

Kemal Ebcioglu

Prefetch Mechanisms by Application Memory Access Pattern

by

Kartik Kandadai Agaram, B.E.; M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2007

Acknowledgments

"Success is the ability to go from one failure to another with no loss of enthusiasm."

–Winston Churchill

It takes many to sustain enthusiasm for so long. I am deeply indebted to my mentor, advisor, and guide, Professor Stephen W. Keckler. From my first day at UT, Steve has provided constant encouragement along with the occasional push when I was distracted or derailed. I would like to follow the example he sets everyday; I am not there yet.

Great thanks go to my collaborators. Professor Kathryn S. McKinley broadened my horizons immeasurably by always having the right paper for me to read at each point in my journey. Dr. Kemal Ebcioglu taught me much about rigorous thinking over numerous hours at the whiteboard; later he showed faith in me when I had none. Professors Doug Burger and Calvin Lin gave generously of their expertise over numerous meetings. In particular, I take from them lessons on the importance of maintaining a consistent interface to the world, and on the craft of writing clearly.

Conversations were one of my constant delights over the years of graduate school. My colleagues in the CART lab were responsible for most of them, and I am indebted to them for their constant feedback on my writing and my presentation. M. S. Hrishikesh and Simha Sethumadhavan tolerated

me as a roommate; they contributed greatly to the immersiveness of my graduate school experience. Heather Hanson was always there when I needed a hand, and I appreciate our many heart-to-hearts about the Ph.D. process. I learnt much about the craft of critiquing and evaluating research by practicing opinions around Karu Sankaralingam, Vikas Agarwal, Sadia Sharif, Ramadass Nagarajan, Paul Gratz, Boris Grot, Changkyu Kim, Raj Desikan, Madhu Saravana Sibi Govindan, Suriya Subramanian, Nitya Ranganathan, and Madhavi Krishnan. The opinions I have left today are the ones that survived them.

A research group is only as effective as the department and support structure it is immersed in. I would like to thank Gem Naivar, Fletcher Mattox, and the rest of the staff at UTCS for shielding us from the inevitable bureaucracy, and for being there for us above and beyond any reasonable expectation. I am very grateful to all the Faculty members I did my coursework with. In particular, Professor Gordon Novak's course on AI in my very first semester at UT left a deep impression on me, as did Professor J Moore's course on the relationship between recursion and induction. In my time here I had also the pleasure to watch a public but friendly duel on religion by Professors Benjamin Kuipers and Raymond Mooney; it caused me to start thinking about the broader context that research is done in within a society. The amazing libraries here at UT helped to fuel that spark.

The graduate students in the broader department helped to leaven and broaden my years here. Maria Jump was always around in the early mornings; on the few occasions I was up at that time her presence never failed to make

things better. In addition to all her other feedback and support, I will always be grateful to Alison Norman for a certain phone call at just the right moment. I wouldn't be here without her. The social life in the department was a rich source of sustenance, especially with Michael Bond, Jennifer Sartor, Serita Nelesen, Nedialko Dimitrov, Walter Chang, Benjamin Wiedermann, and Milind Kulkarni. Above all, I have been fortunate to have found the best of friends during my stay in Austin, especially Markus Fitza, Rocio Ocon-Garrido, Vijay Subramaniam, Jyothy Potluri, Reetu Naik, Varun Mehta, Neha Verma, Julia Kays, Kavita Agrawal, Ashwini Gopal, Anupama Madabhushi, and Uma Bhat. You have helped more than you know.

The enthusiasm everyone here has helped to sustain was created in the bosom of a large and very comfortable family. My parents, Mythili and A. K. Srinivasan, influenced me in ways no child could have noticed; I constantly discover new details of subtlety in the wisdom with which they raised me. My brother, Srikanth Agaram, shared my formative years. His reticent presence was immense. My sister, Vinodhini Krishnan, has seen me at my best and my worst, and loved me yet. As I grow up my relationship with them all has matured into an irreplaceable source of conversation and more. My grandfather, A. K. Rangaraj, was always around in spirit, especially everytime I ran into a book of his around the house. My grandmothers, Pushpa Venugopal and Anusuya Rangaraj, were a precious source of support in an otherwise very male household. These are but the tip of the iceberg; I have been fortunate at every stage of my life to have had some warm source of family around me. In particular, everybody at 45 who watched over me as I grew up over the years, and my parents away from home: Vimala and S. Krishnan, Vijayalakshmi and V. Gopalan, Samyukta and Ramprasad Kandadai, and Kamala and A. K. Sampath. The stressful times when I feel out on a limb are sustained by the years of shelter provided by you all.

I have always needed role models. Before them all was my grandfather, K. Venugopal. I have always looked up to him and marvelled at his strength. When I could finally match him for height I realized I had matched but the most superficial of his attributes. I never had the honour to meet Professor Edsger W. Dijkstra. He sits at my shoulder when I program.

Writing these acknowledgments has been a surprisingly valuable opportunity to take stock of the lessons I have learnt over the years. For this I am doubly grateful to you all. My accomplishments are all yours; I have been but a catalyst.

Prefetch Mechanisms by Application Memory Access Pattern

Publication No. ____________

Kartik Kandadai Agaram, Ph.D.
The University of Texas at Austin, 2007

Supervisor: Stephen W. Keckler

Modern computer systems spend a substantial fraction of their running time waiting for data from memory. While prefetching has been a promising avenue of research for reducing and tolerating latencies to memory, it has also been a challenge to implement. This challenge exists largely because of the growing complexity of memory hierarchies and the wide variety of application behaviors. In this dissertation we propose a new methodology that emphasizes decomposing complex behavior at the application level into regular components that are intelligible at a high level to the architect.

This dissertation is divided into three stages. In the first, we build tools to help decompose application behavior by data structure and phase, and use these tools to create a richer picture of application behavior than with conventional simulation tools, yielding compressed summaries of dominant access patterns. The variety of access patterns drives the next stage: design of a prefetch system that improves on the state of the art.

Every prefetching system must make low-overhead decisions on what to prefetch, when to prefetch it, and where to store prefetched data. Visualizing application access patterns allows us to articulate the subtleties in making these decisions and the many ways that a mechanism that improves one decision for one set of applications may degrade the quality of another decision for a different set. Our insights lead us to a new system called *TwoStep* with a small set of independent but synergistic mechanisms.

In the third stage we perform a detailed evaluation of TwoStep. We find that while it outperforms past approaches for the most irregular applications in our benchmark suite, it is unable to improve on the speedups for more regular applications. Understanding why leads to an improved understanding of two general categories of prefetch techniques. Prefetching can either look back at past history or look forward by precomputing an application's future requirements. Applications with a low compute-access ratio can benefit from history-based prefetching *if* their access pattern is not too irregular. Applications with irregular access patterns may benefit from precomputation-based prefetching, *as long as* their compute-access ratio is not too low.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

For about two decades starting in the early '80s, processor clock speed improved by approximately 50% per year, while DRAM speed only improved at about 7% per year. As a result, the speed gap between processor and main memory cycle time doubled approximately every 6.2 years [8, 32]. Processor speeds have since largely stopped their exponential growth, but modern systems must still deal with latencies to main memory of up to 2000 cycles.

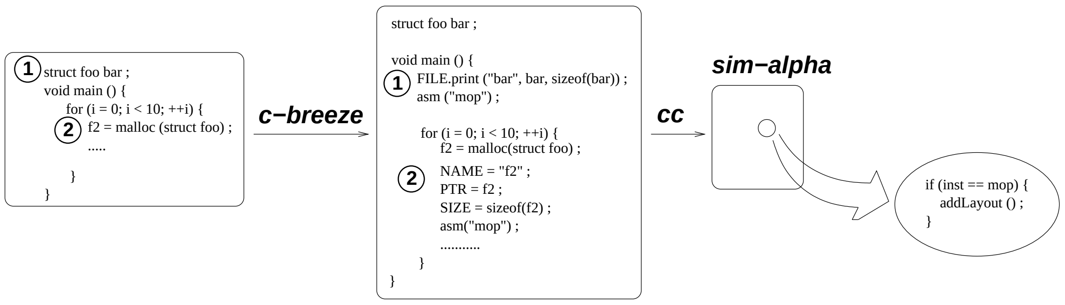
Cache hierarchies have grown in importance as a way to mitigate the effects of this speed gap [46, 73, 87–89]; today's microprocessors often have three levels of cache memories, with each level filtering the address stream seen by lower levels. Caches however make assumptions of spatial and temporal memory locality that are not always valid, and many programs still spend a substantial fraction of their time stalling for memory.

The problem of increasing memory latency has consumed much research effort, and yielded significant new advances. Prior work in memory-system may be categorized into two classes: latency avoidance, and latency tolerance. Latency avoidance techniques attempt to reduce average memory access time (AMAT) for a set of common access patterns. Such techniques include among others multi-word cache-lines to exploit spatial locality, victim buffers, and skewed-associative caches to mitigate conflict misses [42, 80].

Latency tolerance techniques try to find independent useful work to do while they wait for long-latency memory access to complete. Examples of latency tolerance are pipelined memories and banked structures that can be accessed in parallel [15], out-of-order processors and non-blocking caches to discover local parallelism in a serial representation of software [4, 18, 39], and more global uses of parallelism such as multi-threading [48].

Prior work has also emphasized a specific sub-category of latency tolerance technique. Scheduling techniques attempt to neutralize AMAT by using the various levels of the memory hierarchy as staging stations for the effective transfer of useful data to the processor. They include instruction scheduling for load latencies and software-pipelining in the compiler [58, 65], scheduling accesses to DRAM in hardware [37], a variety of prefetch techniques in software and hardware, and techniques such as read-miss clustering [69].

In spite of these advances, the memory system continues to be a major bottleneck to performance while the variety of applications has continued to grow. While the above techniques are often effective, their effect varies for different applications, and it is hard to estimate *a priori* the interaction between a specific class of optimizations and a specific application. As applications and the systems they run on grow more complex, it becomes more difficult to determine potential sources of inefficiency and mismatch between the two. Given the requirement to handle a variety of application workloads, scheduling



```
(1) struct foo bar ;
    void main () {
        for (i = 0; i < 10; ++i) {
  (2)       f2 = malloc (struct foo) ;
            .....

        }
    }
```

```
struct foo bar ;

void main () {
(1) FILE.print ("bar", bar, sizeof(bar)) ;
    asm ("mop") ;

    for (i = 0; i < 10; ++i) {
        f2 = malloc(struct foo) ;
  (2)   NAME = "f2" ;
        PTR = f2 ;
        SIZE = sizeof(f2) ;
        asm("mop") ;
        ...........
    }
}
```

```
if (inst == mop) {
    addLayout () ;
}
```

Figure 1.1: DTrack toolchain

promises the greatest flexibility at runtime in adapting to the needs of different programs without dilating critical paths in a memory access.

In this dissertation we perform a detailed application characterization that decomposes program behavior by data structure and phase. We summarize the rich picture provided by such data into dominant access patterns for different phases in each application. We then focus on the irregular applications that are challenging for prior work and describe some key properties of these programs. These key properties then drive the next phase: the design of a novel prefetching microarchitecture called TwoStep. The rest of this chapter outlines this process in greater detail.

1.1 Detailed application characterization: data structures and phases

Understanding how applications use the memory system is important to at least three groups: (1) system designers who can apply insights into memory system usage to improve hardware and software memory optimization techniques, (2) application writers who can understand how their program uses the memory system and optimize for better locality, and (3) benchmark developers who want to ensure that the diverse patterns of behavior in realistic applications are represented. While many tools have been developed to analyze memory behavior [53, 60, 63, 96], none give insight into the behavior of individual data structures within a program. Our tool — *DTrack* — gathers memory system statistics on a per data structure basis, to help identify those data structures that have the strongest influence on performance and to offer insight into their size and access patterns.

Figure 1.1 outlines the structure of the DTrack toolchain. DTrack consists of a C-to-C compiler that automatically instruments variable allocations in programs and a detailed timing simulator that consumes this instrumentation. This combination yields a tool that generates data profiles - detailed breakdowns of cache misses by the different high-level data structures in the source code. In our experiments with DTrack, we measure the distribution of misses in major data structures, the impact of these misses on total cycle count and on time spent stalling in the pipeline.

Given this data profile, we then manually combine it with a conventional code profile to determine the dominant access patterns for each data structure. Figure 1.2 summarizes the access patterns of three representative applications as the manner in which the major loops traverse the major data structures. Since most cache misses in these programs occur within these loops, we can focus on them and treat the entire application as simply a sequence of

I. **179.art**

```
i = i+1 {
    f1[i]
}
```

a)

```
i = i+1 {
    bu[i]
}
```

b)

II. **181.mcf**

```
i = i+1 {
    node[i]
}
```

a)

```
node = DFS(node) {
    node->child
    node->parent
    node->sibling
    node->prevSibling
}
```

b)

III. **300.twolf**

```
i = rand() {
    t1 = b[c[i]->cblock]
    t2 = t1->tile->term
    t3 = n[t2->net]
}
```

Figure 1.2: Access patterns of major loops: the sequence of objects touched in each iteration. The expression outside the body shows how the induction variable changes for each loop (DFS denotes depth-first traversal); the body enumerates important loads dependent on the induction variable.

iterations from its major loops. The major loops in all our applications have the following key properties:

- They exhibit a wide variety of access patterns, both between different applications and within some applications.
- While access patterns can be very different in different loops, each loop can be summarized in a symbolic manner like the examples in the previous section.
- Each loop iteration performs a series of memory accesses that are often chained together by data and control dependences.
- Even though individual loop footprints can far exceed conventional cache capacities, the footprint of each individual loop iteration is small and occupies just a few cachelines of a normal level-1 data (DL1) cache.
- Most of the hard cache misses occur on the first access to an object in a loop iteration.

All but the last of these points are conventional wisdom; our characterization helped us to quantify their effects, and to focus our attention on *these* particular properties.

1.2 Summary of prior approaches

Numerous prefetching techniques have been proposed in the literature, using both software and hardware, and initiating both single short-range prefetches and long-range sequences of prefetches at a time. Purely *software prefetching*, using the compiler to strategically place prefetch instructions in an application's instruction stream, is a common approach [13, 59]. However, it is often hard for the compiler to statically place a prefetch the right distance before its use. If the prefetch is too close to its use, its latency is not entirely overlapped; if the prefetch is too far, the prefetch is likely to pollute the cache and itself be evicted before use.

Prefetching with hardware support provides greater flexibility at run-time in modulating the slack between prefetch and use based on application needs. Prior studies have resulted in many such prefetching techniques, first issuing prefetches one at a time, either under compiler control [13, 59] or using special hardware that is triggered on specific events such as cache accesses [87], cache misses [16, 41] and dead block speculation [49].

Under the pressure of growing latencies to main memory, recent work has focussed on ways to issue systems of prefetches at a time. The search for ways to determine sequences of addresses to prefetch has proceeded in two largely independent directions driven by conflicting application requirements. The first consists of using prior history in an application's execution to speculatively select systems of prefetches, expressed either as a region of the address space [55, 99] or as an affine function [43, 85].

The second direction consists of precomputation - creating a prefetch thread in either hardware or software that runs ahead of the application and determines what to prefetch [11, 66, 93, 103, 105]. This precomputation may come from running special kernel programs, copies of the application under various speculative modes, or dynamically generated sequences of instructions. Both approaches have drawbacks. History-based approaches are unable to generate accurate prefetches in the presence of arbitrarily complex access patterns. On the other hand, open problems in precomputation-based approaches are low-overhead throttling to avoid cache pollution when the prefetch thread runs too far ahead, and prioritizing between independent prefetches issued by the prefetch thread.

Summary of drawbacks: The state of the art in prefetch techniques has several major limitations; the major decisions of what to prefetch, when to prefetch it and where to prefetch to remain challenges in their most general setting. First, deciding what to prefetch is a challenge for irregular programs that interleave spatial access and pointer dereference in complex ways, and modern prefetch techniques are often better tuned for one of those access patterns than others, such as prefetching arrays or chasing pointers. Applications whose access patterns are too complex for current approaches are also often the ones with the worst baseline performance and therefore most in need of improvement. They are also unlikely to fade in importance; current trends of growing application footprint, increasing software complexity and the need for greater flexibility at deployment-time have made the use of pointers increasingly common [10, 67, 72].

Second, mechanisms that improve prefetch accuracy for one set of appli-

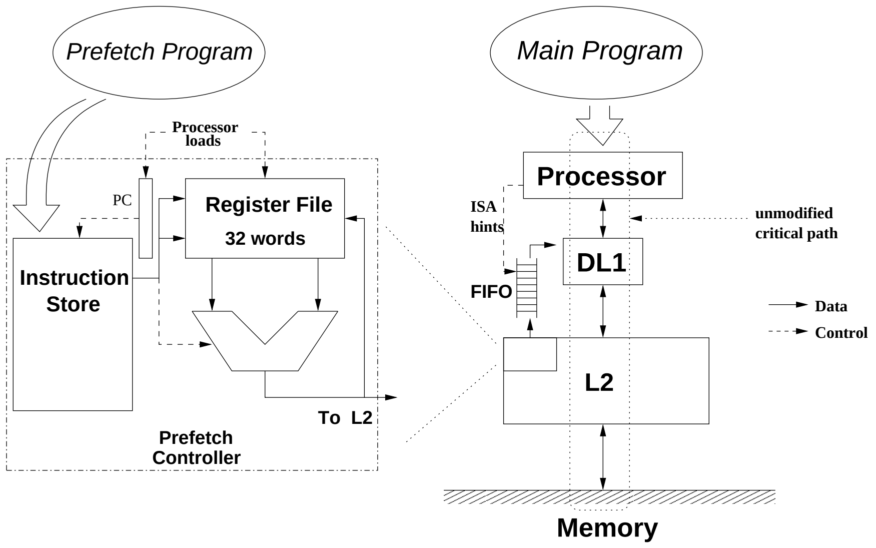


Figure 1.3: The TwoStep prefetching system

cations often end up causing tighter timing constraints for another set resulting in prefetches that are either initiated too late to be effective or those that enter the cache too early and pollute it.

Finally, the greater sensitivity of the DL1 to pollution has resulted in most approaches prefetching exclusively to the L2. We now outline our approach to address these drawbacks, driving our study with a detailed characterization of application characteristics.

1.3 TwoStep: Microarchitecture and compiler for precomputation-based prefetching

The competing advantages of history- and precomputation-based prefetching are largely complementary. Rather than choose between the two, we call for a synthesis driven by application characteristics. Our approach is to select between history- or precomputation-based prefetching depending on whether the application is respectively more likely to be constrained by MLP or prefetch accuracy, using the twin metrics of computation per memory access and access-pattern irregularity. Our results show that these metrics are effective at predicting which applications will benefit from history-based and which from precomputation-based prefetching.

We begin our design by focussing on the challenges posed by irregular programs and use the above analysis to guide the design of a novel precomputation-based prefetching system - *TwoStep*. Our design (Figure 1.3) consists of a statically-generated prefetch program that executes on a programmable prefetch controller. Our prefetch programs are powerful enough to encapsulate strided, pointer and index-array access. This allows us to cover the broad variety of access patterns. In order to minimize latency between dependent prefetches, we place the prefetch controller in the L2. In order to avoid pollution in the DL1 we push each prefetch from L2 to a FIFO between L2 and DL1. Prefetch culminates in the movement of a fixed number of cache-lines into the level-1 data (DL1) cache. Since the focus of TwoStep prefetching is on the first access to each object in a loop, this movement is orchestrated by an ISA enhancement we call the `Pull` instruction, inserted at the start of each loop iteration in order to bring into the DL1 the cache-lines that constitute the working set of that iteration. Since loop iteration footprint is low, pollution in the DL1 due to occasionally inaccurate prefetches is bounded. Finally, the presence of the FIFO and `Pull` instructions makes it easy to throttle the prefetch thread — the prefetch program stalls when the FIFO is full. This lightweight mechanism for throttling avoids polluting the L2.

We implement a compiler for TwoStep to automate the generation of prefetch kernels from application source code. Our compiler improves on the state of the art [47] by requiring less profile information (iteration counts for loops only) and by performing a more aggressive search of the state space of loop cluster combinations to select the most favorable loops. The combination of compiler support and these microarchitectural mechanisms provides effective prefetching for irregular applications, including several that have been challenging to prior work.

However, comparisons with Guided Region Prefetching [99] show that precomputation fails to achieve as much benefit on more regular applications with spatial locality. A detailed analysis reveals that the trends shown by the two competing techniques are representative of the more general classes they belong to: backward-looking history-based prefetching vs forward-looking precomputation-based prefetching. History-based prefetching consists of tracking the history of the address stream for an application and making predictions based on the assumption that future behavior will be similar to the past. Precomputation-based prefetching, on the other hand, does not make this assumption and instead explicitly precomputes the application's future needs.

We find that application affinity for one class or the other is decided by two major properties: access pattern regularity and computation per memory access. Applications with irregular access patterns will clearly have high affinities for history-based prefetching. This is not surprising; regular access patterns are easier to predict based on knowledge of the past address stream. Conversely, we expect irregular applications to prefer precomputation-based prefetching. More surprising, however, applications with irregular access patterns require more computation per memory access in order to benefit from precomputation-based prefetching. The greater prevalence of dependences and sequentialization causes poor utilization of prefetch bandwidth and makes them more sensitive to the critical path in a loop.

When the memory footprint of a loop exhibits significant locality, history-based prefetching can issue prefetches in parallel and tolerate much 'tighter' loops with less computation per memory access. However, such approaches fail to benefit applications with low spatial locality, and accurate prefetching requires a precomputation thread to run ahead of the main program generating prefetches. This approach is however constrained in its memory-level parallelism, and as a result cannot be applied to loops with low levels of computation per memory access. This analysis of the state space provides the basic intuition behind the complementary nature of these two categories of prefetching. Different loops in an application require either one or the other. As a result, combining region prefetching with precomputation is a feasible approach, and we show that this combination successfully achieves the best of both worlds.

1.4 Dissertation organization and contributions

In this dissertation we focus on the shortcomings of past work in prefetching irregular memory-intensive applications and try to remedy these shortcomings without compromising hard-won improvements for other applications. Our solution combines features from software and hardware as well as local and global approaches to prefetching. It consists of a compiler-generated prefetch program that runs on a simple in-order programmable prefetch controller in the level-2 cache (L2) [99]; a FIFO between the L2 and the level-1 data (DL1) cache that receives every prefetch generated by the prefetch controller [97]; and ISA enhancements that provide hints on each loop iteration in the main program, including its bounds, expected footprint, and access patterns. The ISA enhancements encode general properties about a program that could be used by other techniques as well, and we show how to use them to orchestrate data transfer from FIFO to DL1. In particular, this thesis makes three contributions:

- A detailed characterization of irregular applications to first establish the feasibility of overlapping access latency in them, and then glean some insight into their access patterns.
- The design and evaluation of a prefetch technique called *TwoStep* that combines the benefits of software and hardware as well as short- and long-range prefetching.

- The insight that precomputation- and history-based prefetching are complementary approaches, with strengths and weaknesses in opposition to each other. Applications with irregular access patterns can benefit from the the greater flexibility of precomputation; applications with low computation per memory access require the better bandwidth efficiency of history-based approaches.

The rest of this thesis is structured as follows. In Chapter 2 we survey the prior literature in several areas related to this dissertation. Chapter 3 describes our framework for decomposing memory behavior by data structure and summarizes the results of this study. Chapter 4 similarly describes our framework for studying phase behavior, with a novel adaptive algorithm to identify the best granularity at which to view the phase behavior of an application. Chapter 5 describes our TwoStep prefetch microarchitecture and presents the results of an initial study with hand-crafted kernels. Chapter 6 describes the TwoStep compiler and characterizes the state space seen by it for our applications. Chapter 7 puts microarchitecture and compiler together for a comprehensive evaluation, quantifying the strengths and weaknesses of TwoStep compared to other techniques that rely on spatial locality, and showing that the two kinds of prefetching are amenable to recombination. Finally, Chapter 8 summarizes our insights from this work and identifies areas for future study.

Chapter 2

Background and related work

In this section, we summarize the related work that we build upon in this thesis. In tune with the structure of the thesis, we break down our analysis into three categories - memory visualization and characterization tools relevant to DTrack, the body of prefetching studies relevant to TwoStep, and finally the prior work in whole-program analysis, pointer analysis and slicing that the TwoStep compiler is based on.

2.1 Visualizing application memory behavior

Simulation is a common method of producing aggregate memory statistics [1, 9, 33, 89]. More sophisticated cache memory behavior analysis tools have been developed [53, 60, 61, 63, 64, 96], and this section compares DTrack to this prior work. Our work differs from these tools in that we consider pointer data structures in addition to arrays, and show that aggregate statistics obscure possible optimization opportunities revealed by phase behavior. This increased detail comes at a cost of increased simulation time.

Most tools have focused on aggregate data structure and procedure-level information for arrays [53, 60, 61]. Lebeck et al. [53] and Martonosi et al. [60] present data structure and procedure level aggregate miss information, and classify misses as compulsory, capacity, and conflict. Both papers also present a number of software optimizations for improving cache performance. While these tools point users to the code and arrays that cause problems, they examine the behavior of an array within the context of a single procedure, resulting in two weaknesses. First, because they do not perform cross data structure analysis, it is not directly apparent from their aggregate data statistics which data structures interfere with themselves or with others. Second, since they do not perform cross-procedure analysis, optimizations chosen to improve performance of one array/procedure combination may diminish performance in another procedure. Finally, both tools handle only regular array-based data structures rather than pointer-based data structures. McKinley and Temam analyze the complementary dimension of inter-nest and intra-nest loop locality [63, 64], but again consider only arrays and aggregate information between loop nests.

2.2 Analyzing time-varying behavior

Several tools have studied time-varying behavior. The Cache Visualization Tool [96] demonstrates the time-varying behavior of arrays as they march through the cache. The graphical component of this tool colors cache-lines according to their locality and misses by data structures, so the user can see which cache-lines cause conflict misses. This level of detail supports analyzing a single loop nest at a time, whereas we analyze data structure phase behavior across much longer periods. Chilimbi et al. [20, 78] analyze compressed program traces, decompose them into *hot data streams*, and use these hot data streams to drive layout and prefetching optimizations. This approach of searching for access patterns across the different data structures in a program is complementary to ours, which attempts to decompose application access patterns by data structure. We believe our approach is more effective at providing intuitions about application behavior that are useful to humans in different roles.

More recently, several studies have used some form of code signature to detect phase boundaries. Basic Block Vectors (BBVs) are currently the most accurate method to generate code signatures, and several studies explore their uses in clustering phases and detecting phase transitions in an offline [83, 84] and online [86] setting. One alternative to BBVs is the use of program counter or Extended Instruction Pointer Vectors (EIPVs) [6], whose merits have been debated by Lau et al. [51]. Another alternative consists of more high-level metrics based on code structure, such as register use vectors or loop vectors [52]. All these studies, however, select an arbitrary sampling period and use it for all the applications they evaluate. In this study, we provide a more rigorous method to separately determine the correct sampling period for each application.

Perhaps the most similar work to ours is the online phase detector of Nagpurkar et al. [68]. Their system maintains a current window of object references within a JVM and assesses the similarity of the recent references in it to those in an older trailing window. Like our study they evaluate the effect of window size (sampling interval) on phase detection. While our study looks for phases in fine-grained behavioral statistics of an application, they study phase behavior in the functional list of object references touched by an application. The two approaches are complementary.

2.3 Prefetching

Prefetching has been an important tool in combating growing memory latencies in both the compiler and microarchitecture, and as a result there is a large body of research in this area. We break it down into several categories below, focussing on important studies in each and elaborating on their relationship with our scheme.

Spatial prefetching and stream buffers: The earliest systems performed prefetching for array-based numerical codes. Software-based solutions detected array references and loop induction variables to prefetch a fixed number of iterations in advance for complex loop nests [12, 59]. These solutions were geared towards array-based applications with a very different patterns of behavior from our focus in this work, and we do not consider them further. The earliest hardware prefetch systems systems simply brought in the next cache-line on a miss [87]. Developments and enhancements have proceeded along several directions. First, a variety of techniques have been studied for region prefetching, culminating in the work of Lin et al. [55]. Second, spatial hardware prefetchers used stream buffers to avoid cache pollution in the presence of inaccurate prefetches [42, 45, 71]; we focus on two exemplars of the state of the art. Sherwood and Calder [85] couple stride prediction with stream buffers, while Hur and Lin [38] adaptively vary stream length at an application granularity. Our mechanism draws inspiration from stream buffers as a mechanism to avoid cache pollution. However, stream buffers are inadequate to our needs for two reasons. First, they lengthen the critical path of a normal cache access to search a cache and associated stream buffers, either in series or parallel. Secondly, the stream-buffer approach to handling inaccuracies in prediction does not fit our model. Stream buffers can be seen as a constantly evolving set of hypotheses on the stream of addresses that a program needs. When one fails, the stream buffer is simply flushed to make way for another hypothesis. In the context of irregular applications, however, the compiler-supplied hypothesis is a valuable resource and our mechanism is able to tolerate momentary inaccuracies in the FIFO without needing to frequently flush it. While Hur and Lin do not spend time constructing elaborate hypotheses, their approach focusses exclusively on spatial cache misses, finding short streams even in irregular programs. Our approach is complementary, focussing instead on the more difficult non-spatial cache misses.

Software prefetching by compiler-inserted instructions: Based on earlier work on array-based programs, Lipasti et al. performed an early study showing that benefits could be obtained by prefetching pointers passed as parameters to function calls [56]. Luk and Mowry identified the main problem to overcome in array-based prefetching: the presence of pointers introduces a serialization between prefetches, so that prior prefetches must return before more progress can be made [59]. They performed a thorough analysis of the use of jump pointers to overcome this serialization. Cahoon and McKinley built on the work of Luk and Mowry by performing interprocedural dataflow analysis in an object-oriented environment with virtual-method calls [12]. These studies handled regular pointer-based codes such as linked-list and binary tree traversal with success. However they are unable to adapt the slack given to prefetches at runtime.

Hardware prefetching by detecting patterns in the address stream: Another line of prefetching studies add hardware enhancements to support the prefetching decision. A number of studies have found successively more sophisticated patterns to prefetch by observing the patterns of an application's address stream. We note the progression of ideas from early studies on detecting variable-stride patterns such as by Chen and Baer [19], through studies on Markov prefetchers that use cache misses to trigger further cache accesses [5, 41, 77], finally culminating in the work of Iacobovici et al. [40], which presents complex stride-detection hardware to track and predict a variety of affine access patterns. Dead-block correlating prefetchers are another development on this idea, triggering prefetches not on specific cache misses, but on the earlier speculative eviction of cache-lines [49]. All these studies assume that there are patterns to be found in the address trace, and in practice are at the mercy of pathologies of memory allocators. They also need cache misses to perform prefetches, and are therefore self-limiting in the improvement they can bring.

Hardware-based pointer prefetching: Several studies have attempted to model pointers themselves rather than raw address streams. An early exponent was the study of jump pointers by Roth and Sohi [76], showing them to be feasible for prefetching in both software using hand-coded kernels and in hardware using a specialized unit to construct chains of jump pointers and store them in the interstices of heap allocations. In spite of being amenable to implementation in hardware, jump pointer-based prefetching suffers from the classic problem of software prefetching - an inability to adaptively time prefetches based on dynamic changes to a program.

Recent work on content-directed prefetching emphasizes this aspect [3, 23]. These studies contain a prefetch mechanism consisting of a simple hardware unit that scans incoming cache-lines for pointers and initiates prefetches along them. They also include a reinforcement mechanism that adaptively prunes pointer paths that a program does not use. This approach has two drawbacks. First, it addresses pointer and indirect prefetches, but is unable to avoid spatial misses for objects larger than a cache-line. TwoStep is able to handle arbitrary interleavings of regular and irregular types of access. Second, like address-stream-based approaches described above, it relies on cache misses to trigger prefetches albeit in a more efficient manner. TwoStep allows the prefetch thread the opportunity to run ahead regardless of cache misses or other pipeline state. As an extreme example, a low-ILP application with a high computate-store ratio but irregular access patterns would spend a significant portion of its time stalling for memory in spite of such a pointer prefetch system. TwoStep would however be able to stay ahead of the main program and avoid most DL1 misses.

Programmable prefetch engines: While the above pointer prefetching studies could get multiple iterations ahead of the main program, they were focussed on pointers alone and unable to handle more sophisticated access patterns combining spatial and pointer access. A couple of recent studies have addressed this. Guided Region Prefetching by Wang et al. provides hints in load instructions that can permit the L2-based prefetch engine to run ahead of the program [99]. However, this work avoids pollution by a hard bound on the number of iterations the prefetcher can run ahead. The Push model of Yang et al. adds engines at each level in the cache hierarchy that each execute specialized kernels to push data to the level above [102]. Compared to our work, that study has several differences. First, it is designed for purely pointer-based traversals and is unable to handle combinations of spatial and pointer-based access. Second, it involves much more hardware complexity by adding engines at each level of the memory hierarchy, engines that are superscalar and implement complex heuristics for prioritizing and throttling accesses. The use of a FIFO serves to substantially simplify our design relative to theirs.

A third study with some similarity to our own is the programmable prefetch engine of VanderWiel and Lilja [97]. This study uses a prefetch engine similar to ours that prefetches to both DL1 and L2. However, it avoids pollution by using tags on cache-lines (rather than on instructions as in TwoStep) to maintain a producer-consumer relationship between processor and prefetch engine. In spite of being programmable, this engine was designed for largely array-based codes, and used a simple intra-procedural analysis to generate prefetch programs. TwoStep extends this approach to support irregular applications.

Novel processor architectures with prefetching effects: The primary architectural idea inspiring TwoStep was the decoupled access/execute architecture of Smith [90]. We believe it is the work closest in spirit to ours, using software-controlled queues to manage slip between execution and memory-access "streams". Designed in a very different context, the motivation of this design was to sidestep the Flynn bottleneck (approximating later superscalar designs) and to overlap multiple instructions with simple issue logic (approximating out-of-order execution). It is useful to enumerate the differences between decoupled architecture and TwoStep. Compared to this early study, we maintain an asymmetry between the two streams, relegating the access stream to a purely performance-enhancing function and reducing the frequency of synchronization “handshakes” between the two streams.

Several recent studies have made dramatic changes in overall processor microarchitecture, resulting in prefetching effects among other benefits. The RAW architecture reports substantial speedups for irregular applications using a more explicit orchestration of data movement and with loss of compatibility with existing programming models [94]. Over common applications - mcf and twolf - we show comparable improvements in TwoStep but with a more conventional ISA and software stack. Datascalar and Slipstream processors simultaneously run a program on multiple processors and cause it to speed up on each of them [11, 93]. Runahead execution is more parsimonious and utilizes processor resources to run in “speculative” mode when it would otherwise be stalled [66]. While runahead execution has benefits beyond just prefetching, we note that like some of the hardware prefetch schemes above it only performs prefetches during cache misses, thereby being less efficient in overlapping latency. It is also unlikely to be effective in prefetching serialized pointers since a stall in one pointer would invalidate all computations based on it.

Summary: As the above survey shows, TwoStep benefits from the lessons of a large number of prior studies. Many of these studies share some points of similarity but make design decisions that cause them to be ineffective on irregular programs. The novel architectures surveyed above yield some of the benefits of TwoStep but at greater cost or with a change in programming model. A common thread among many prior studies is to use cache miss events to trigger prefetches. Like the designers of dead-block correlation prefetching, we find this approach to be self-limiting [49].

2.4 Slicing and whole-program analysis

Interprocedural or whole-program analysis has been the topic of much research attempting to improve its efficiency in a variety of contexts: programming languages with and without pointers [34, 35], automatic parallelization [79], and a variety of specific analyses such as constant propagation [29], side-effect analysis [21] and escape analysis [7, 26]. Whole-program analysis and pointer analysis often have a symbiotic relationship in the context of languages with pointers like C [17]; aggressive pointer analysis must necessarily be a whole-program analysis, while other applications of whole-program analysis often require points-to information. Again, much effort has been expended on the development of efficient algorithms for whole-program pointer analysis [24, 27].

There has been relatively less work in slicing, with applications largely in the field of program-understanding [36, 50, 100]. Our application of slicing is rather different from this conventional use; while most slicing studies focus on finding minimal slices while retaining full coverage, our focus is on finding sparse regions in a slice that maximize the amount of computation *not* in the slice. In particular, full coverage for pathological cases is not a concern since we use slices for performance, not correctness. Also, while most slicing studies use a static representation of program structure, simply returning the set of static program statements that belong in a slice, our view is more ordered and context-sensitive: the compiler must return a context-sensitive sequence of statement *instances*.

2.5 Compiler support for precomputation

Compilers for precomputation are based on program slicing and typically operate either by post-compilation binary translation [54, 76, 77] or at runtime in a dynamic compiler [104]. Computing slices in hardware restricts the scope of individual slices, while binary translation detects only simple pointer-chasing patterns. The state of the art in thorough compiler-based precomputation is the work of Kim and Yeung [47]. Kim and Yeung's compiler framework uses 2 kinds of profile information — loop iteration count profiles and cache miss profiles — to select compute precomputation slices for execution in spare hardware contexts of a simultaneous multithreading (SMT) processor. We perform a more detailed comparison of this compiler with ours in Chapter 6.

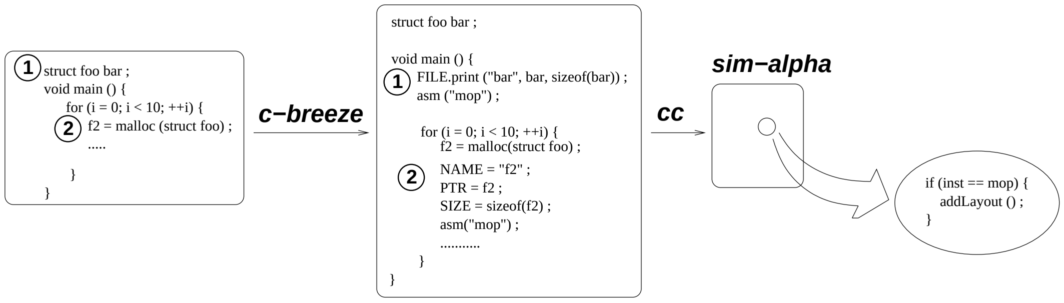
Chapter 3

Data structure decomposition using DTrack

This chapter describes DTrack and our methodology for analyzing applications, and performs a detailed analysis of the data structures of twelve applications. DTrack separates by data structure the stream of addresses an application requests from memory. Our exploration reveals a wide variety of application behaviors and shows that opportunities for overlapping latency exist if hardware can adapt to application requirements.

3.1 DTrack: A tool for studying irregular applications

DTrack consists of a source-transformation tool to automatically instrument memory allocation points in programs and a detailed timing simu-



```
struct foo bar ;
void main () {
    for (i = 0; i < 10; ++i) {
        f2 = malloc (struct foo) ;
        .....

    }
}
```

```
struct foo bar ;

void main () {
    FILE.print ("bar", bar, sizeof(bar)) ;
    asm ("mop") ;

    for (i = 0; i < 10; ++i) {
        f2 = malloc(struct foo) ;
        NAME = "f2" ;
        PTR = f2 ;
        SIZE = sizeof(f2) ;
        asm("mop") ;
        ...........
    }
}
```

```
if (inst == mop) {
    addLayout () ;
}
```

Figure 3.1: DTrack toolchain

lator that consumes this instrumentation. The source instrumentation maps addresses to data structures in order to communicate the address range corresponding to each variable to the simulator. Figure 3.1 shows a schematic of our tool.

The instrumentation tool is an extension to the C-Breeze C-to-C compiler [30], while the simulator is a detailed and validated timing model of the Alpha 21264 pipeline [25]. For each variable in the program, the compiler-generated instrumentation stores the variable's name and address at a designated location in memory and interrupts the simulator by means of a special opcode ("mop" in Figure 3.1). On executing this instruction at runtime, the simulator imports the information from this designated location in simulated memory. Since the simulator knows the extent of each variable in the application at any time, it maps the virtual address of each memory access to a specific variable, and maintains statistics on the progress of the memory access by the data structure it belongs to. Classifying and assigning each load and store to a specific variable slows the simulator down by 60% on average and 100% in the worst case.

3.2 Design decisions

The challenge here is to keep the overhead due to the instrumentation low and to minimize the perturbance to the application. There are two levels of overhead to consider. The first is overhead in the simulator; classifying each load and store to a specific variable and incrementing the appropriate counter slows the simulator down by 60% on average and 100% in the worst case. The second and more serious source of overhead is instrumentation in the application itself. In addition to increasing the simulator's burden, application-level instrumentation could perturb the program under study and so compromise our results. Instrumentation design is therefore guided mainly by minimizing application perturbance:

- Stack variables are not instrumented because the high frequency of scope changes would raise the instrumentation overhead too much. Instead, we treat the stack as a single data structure and coalesce all accesses to it by a simple range test. Our results will show that misses to the stack are generally negligible.

- Global variables have a constant range over the lifetime of an application. We communicate the ranges of these variables by writing them to disk and signalling the simulator as shown by instrumentation "1" in Figure 3.1. Since these file operations are a fixed-time initialization cost, they provide the most efficient amortized mode of communication for global variables.

- Tracking dynamic allocations on the heap is difficult because the same raw address could be allocated to different data structures at different times in a program's execution. DTrack instruments heap allocations and deallocations ("2" in Figure 3.1) and tracks them in the simulator, using them to dynamically change the data structure corresponding to each address. We distinguish data structures on the heap by call-site. As a result we are unable to distinguish between multiple allocations at a single call-site. This design is not a concern in the SPEC-2000 benchmarks we study, but might be a limitation in studying more fine-grained object-oriented applications, where a single allocation site produces lots of objects in multiple data structures.

Taken together, these design decisions are successful at limiting instrumentation overhead to 10 instructions per heap allocation and 4 instructions per deallocation. This results in total overhead of less than 0.6% of total instruction count across all the benchmarks we study except gzip, where the instrumentation is 3.7% of total instruction count because of frequent heap allocations in inner loops.

Alternatives: We considered and discarded several alternatives to this methodology for classifying memory accesses. First, we considered hardware counters rather than simulation to reduce the turn-around time on our results. However, hardware counters do not have the fidelity and flexibility to track cache misses to many specific fine-grained memory regions. Second, we considered using the debugging symbol-table information in application binaries, but we could not find a way to handle applications with custom memory allocators, such as twolf. Our methodology makes it easy to inform the C-Breeze pass about the names and prototypes of application-specific custom allocation routines, along with information about how the size of the allocation is obtained

Feature	Size/Value
Data caches	
DL1 cache	64 KB, blocksize 64 bytes, 2-way, 3 cycles
L2 cache	512 KB, blocksize 64 bytes, direct-mapped, 12 cycles
TLBs	128 entries
Main memory	
Peak bandwidth	1.6Gbytes/s
Rambus geometry	64 banks * 512 rows * 2KB/row
Access latency (cycles)	32 PRER + 24 ACT + 48 RD/WR + queuing
Out-of-order Processor	
Pipeline width	4
Int ALUs, multipliers	4,4
FP ALUs, multipliers	1,1
Branch predictor	Tournament, 1 KB x 1 KB local, 4 KB global, 4 KB choice

Table 3.1: Details of the simulated Alpha 21264-like processor and memory hierarchy

from the arguments to the allocation routine. We began by performing just cache simulation, but migrated to a full-scale timing simulator in order to be able to estimate IPC improvements due to optimizations for specific data structures. Finally, we used a detailed and validated out-of-order processor simulator because Pai et al. showed that an out-of-order processor presents to the memory hierarchy a very different sequence of memory accesses than an in-order processor [70].

Benchmark	IPC	DL1	L2
		Miss-rate	Miss-rate
164.gzip	1.39	2.3	3.9
175.vpr	0.67	3.0	35.3
176.gcc	1.15	3.2	10.4
177.mesa	1.06	0.9	23.4
179.art	0.23	14.8	74.9
181.mcf	0.14	24.1	60.5
183.equake	0.58	14.1	29.4
186.crafty	1.21	1.3	4.3
188.ammp	0.57	10.0	45.0
197.parser	0.97	3.6	21.5
256.bzip2	1.16	2.1	32.6
300.twolf	0.51	9.5	26.9
sphinx	0.58	15.8	41.9

Table 3.2: The benchmarks we use and their aggregate memory hierarchy behavior

3.3 Methodology: Benchmarks, inputs and simulation periods

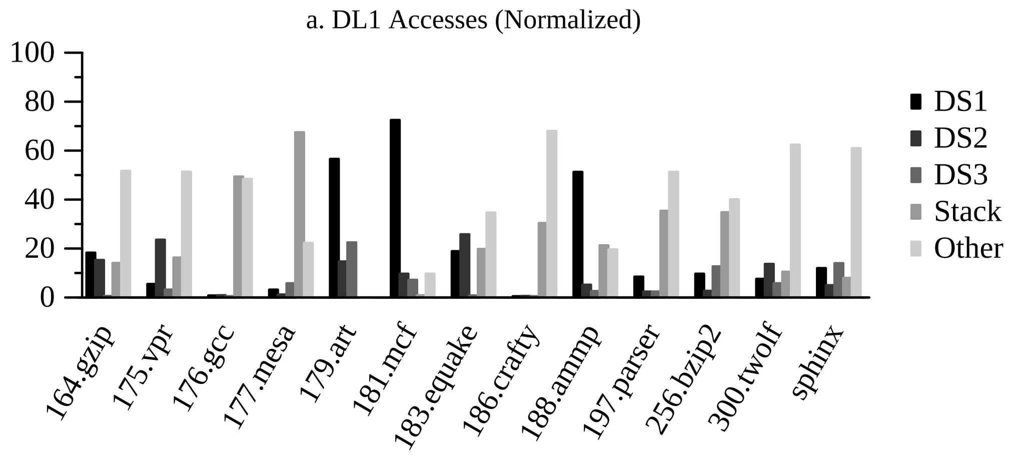
We now describe our methodology for the experiments in this dissertation, including simulated machine configurations, benchmarks and simulation interval selection. We use a version of the sim-alpha [25] timing simulator modified to consume the DTrack instrumentation and maintain cache and TLB statistics by data structure. Figure 3.1 shows the baseline configuration we simulate, including a Rambus memory model. Table 3.2 lists some aggregate properties of the benchmarks we study, including average instructions per cycle (IPC) and miss-rates at the level-1 data (DL1) and level-2 (L2) caches. Our benchmarks range from regular ones such as 179.art to highly irregular ones such as 300.twolf, from compute-bound (164.gzip) to memory-bound (181.mcf). We are unable to study the remaining 3 C benchmarks in the SPEC2000 suite due to methodological difficulties; 253.perlbmk no longer builds on our Alpha platform with the latest version of libc, and 254.gap and 255.vortex run incorrectly on our native Alpha platform because of unaligned addresses generated by their custom memory-managers. While these unaligned addresses could be fixed by modifying the benchmark sources, we estimate that adding the necessary padding could significantly perturb benchmark behavior. All our simulations use the designated ref input set for the corresponding benchmark.

Simulation intervals: We used two sets of simulation intervals for our simulations. First, for the study of global phase behavior in the next chapter we simulated each of our applications to completion. To keep experiment durations reasonable we partitioned the total run-time for each application into chunks of 1 billion instructions, and performed a set of simulations in parallel on a cluster of Linux workstations managed by Condor [57]. Each simulation performs functional simulation for a staggered duration, then performs detailed timing simulation for 1 billion instructions. We then aggregated the results of all these simulations offline to generate phase data for the entire application.

In principle, our parallel approach can introduce errors due to the cold caches that appear every billion instructions. All but one or two billion-instruction samples in each of our benchmarks encounter at least 6.7 million misses in the DL1 and 0.4 million misses in the L2. Only 164.gzip and 177.mesa often have less than 2.8 million L2 misses per billion-instruction sample. Since the error due to extra compulsory misses is a maximum of 512 misses in the DL1 and 8192 misses in the L2 in every billion instructions, the fraction of extra compulsory misses we introduce is no more than 0.05% in the DL1 and 1.8% (0.2% excluding mesa and gzip) in the L2.

The results of these experiments, when correlated with high-level loops, yielded the major outermost loops that constitute more than 90% of the execution of each of our applications. For all our experiments except for phase behavior we then selected one iteration of this outermost loop, demarcating the start and end of this iteration by a special 'marker' opcode using the techniques outlined above, performing fast functional simulation until we reach this opcode, and detailed timing simulation thereafter until reaching the end marker. These simulation periods have been verified to be representative of each application's runtime and aggregate cache miss behavior.

The exceptions to this methodology are the applications 176.gcc, 186.crafty, 197.parser, and sphinx, for which we were unable to generate global phase data due to infrastructural issues. For these applications we determined the end of initialization by inspecting their source code and simulated 500 million instructions past this point.



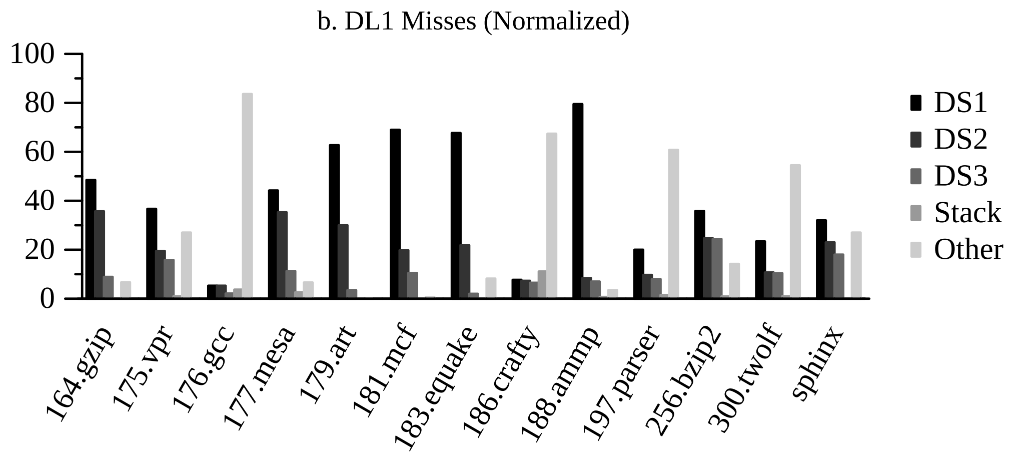


Figure 3.2: Decomposition of DL1 misses and accesses by data structure. L2 misses show similar trends to DL1 misses.

3.4 Results: Data profiles and distributions

Having described DTrack and our experimental methodology, we now present a detailed characterization of the above SPEC benchmarks using DTrack. We begin by studying basic data profiles generated by DTrack, and then explore two ways that this new capability to visualize the behavior of different data structures can be used to help answer sophisticated architectural questions.

DTrack generates data profiles. Figure 3.2 breaks down the aggregate memory behavior of our applications – accesses and miss-rates at the DL1 and L2 – by the three data structures that cause the most DL1 misses (DS1, DS2, DS3), the stack, and everything else. Figure 3.2.a shows that the breakdown of accesses to the DL1 (and therefore the rest of the memory hierarchy) varies greatly across our applications. While 179.art and 181.mcf have skewed distributions, with 80% of all accesses coming from 2 data structures, 300.twolf, 176.gcc and 186.crafty have extremely balanced distributions; no data structure contributes more than 2% of accesses, and it takes 60–100 distinct data structures to account for 90% of cache misses. Other applications lie between these extremes.

While accesses are often spread out, Figure 3.2.b shows that misses tend to cluster. The top 5 data structures usually contribute more than 90% of all DL1 misses. The exceptions are 176.gcc, 186.crafty, and 197.parser with a long tail of minor data structures that respectively end up accounting for 84%, 67% and 78% of all cache misses. Among the other applications, the

Name	Type	Access	Footprint	Object
164.gzip				
`window`	Array	Regular	64KB	2 bytes
`prev`	Array	Regular	64KB	2 bytes
`inbuf`	Array	Regular	184320KB	1 byte
175.vpr				
`rr_node`	Array	Irregular	10638KB	40 bytes
`rr_heap`	Array	Irregular	6717KB	24 bytes
`rr_node_route_inf`	Array	Irregular	2653KB	16 bytes
176.gcc				
`reg_last_sets`	Array	Irregular	0.5KB	8 bytes
`reg_last_uses`	Array	Irregular	0.5KB	8 bytes
`qty_const_insn`	Array	Irregular	4KB	8 bytes
177.mesa				
`Image Buffer`	Array	Regular	2560KB	2 bytes
`Depth Buffer`	Array	Regular	5120KB	4 bytes
`Vertex Buffer`	Array	Regular	920KB	920KB

Table 3.3: Details for some of the major data structures in Figure 3.2.

major data structures end up partitioning cache misses among themselves in a variety of ways; the top data structure can contribute anywhere between 20 and 80% of total cache misses.

Comparing Figures 3.2.a and 3.2.b, we see that cache misses and accesses are poorly correlated. A few applications such as 179.art and 181.mcf reveal a simple underlying organization with only a few data structures, and misses tracking the distribution of accesses. However, the majority of applications show a well-understood pattern where a data structure receives more accesses than another, yet accounts for fewer misses. As expected, the stack accounts for a significant fraction of accesses without ever presenting a signif-

Name	Type	Access	Footprint	Object
179.art				
`f1_layer`	Array	Regular	625KB	64 bytes
`bus`	Array	Regular	859KB	8 bytes
`tds`	Array	Regular	859KB	8 bytes
181.mcf				
`nodes`	Array	Regular & irregular	7071KB	120 bytes
`arcs`	Array	Irregular	188416KB	64 bytes
`dummy_arcs`	Array	Irregular	3771KB	64 bytes
163.equake				
`K`	3D Array	Regular	22399KB	8 bytes
`disp`	3D Array	Regular	2828KB	8 bytes
`M`	3D Array	Regular	943KB	8 bytes
186.crafty				
`rook_attacks`	Array	Irregular	127KB	8 bytes
`last_ones`	Array	Irregular	64KB	1 byte
`first_ones`	Array	Irregular	64KB	1 byte

Table 3.4: Descriptions of the major data structures in Figure 3.2 (cont'd).

icant problem to the DL1. The sole exception is 186.crafty where the stack collectively contributes more misses than any single global data structure. As we have seen, however, 186.crafty has a very balanced distribution, and the stack still accounts for only 11% of DL1 misses.

3.5 Data structure details

So far we have looked at differences in miss distribution across the major data structures in the different SPEC benchmarks while hiding details about the individual data structures behind the anonymous names DS1, DS2 and

Name	Type	Access	Footprint	Object
188.ammp				
`atoms`	Pointer	Regular & irregular	41322KB	2208 bytes
`nodelist`	Array	Regular	76KB	232 bytes
`atomlist`	Array	Regular	4372KB	232 bytes
197.parser				
`Connector`	Various	Irregular	variable	24 bytes
`Disjunct`	Various	Irregular	variable	40 bytes
`table`	Various	Irregular	variable	40 bytes
256.bzip2				
`block`	Various	Irregular	900KB	1 byte
`quadrant`	Various	Irregular	1800KB	2 bytes
`zptr`	Various	Irregular	3600KB	4 bytes
300.twolf				
`net_array[]`$\rightarrow$`netptr`	Pointer	Irregular	253KB	48 bytes
`tmp_rows`	Array	Irregular	34KB	1 byte
`rows`	Array	Irregular	34KB	1 byte
sphinx				
`Model`	Array	Irregular	3343KB	168 bytes
`hmms`	Array	Irregular	3531KB	76 bytes

Table 3.5: Descriptions of the major data structures in Figure 3.2 (cont'd).

DS3. Tables 3.3–3.5 summarize the high-level details of these data structures. For each benchmark, we show the name of these data structures as used in the source code, along with a brief summary of the type of the data structure (array or recursive), whether it is predominantly accessed in a *regular* fashion with spatial locality or in an irregular fashion with low spatial locality. Finally, we provide the size of each object in these data structures and their total sizes in the application.

The major data structures are predominantly array-based in the applications we study. However, these data structures are often used to emulate complex graphs using either real pointers (181.mcf:nodes, 175.vpr:rr_node) or integers that index into other arrays (256.bzip2:quadrant, 300.twolf:rows). The wide variety of uses indicate that data structures are often declared to be arrays solely to simplify memory management. Most of the major data structures are dynamically allocated on the heap. The major exceptions are 186.crafty that causes a significant fraction of misses to the global segment, and 176.gcc which allocates most of its variables on the stack using `alloca`. We now examine the wide variety of patterns by which these data structures are accessed.

3.6 Data structure access patterns

This detailed decomposition provides a glimpse into the array of behaviors shown by the different data structures in a single application, ranging from uniformly regular or irregular access across all major data structures to a combination of access patterns for different data structures. There is no pattern in fraction of footprint or total accesses that these data structures occupy. A data structure's access and miss rank is often not the same, and the distribution of misses among the major data structures varies widely across applications. Accounting for 90% of DL1 misses requires between 2 and 25 distinct data structures for different programs. Finally, applications where irregular accesses dominate - such as mcf - show synergistic effects between data structures; improving multiple data structures simultaneously does sig-

```
// complex termination condition not shown
loop for cell = carray[$random]:
    if cell->class == -1:
        continue
    blkptr = barray[cell->cblock]       //   8 bytes
    tile = cell->tileptr                //  16 bytes
    term = tile->termsptr               //  64 bytes

    loop 3 times:
        loop until term is null:
            net = term->net
            a = netarray[net]           // 128 bytes
            b = term->termptr           //  64 bytes
            c = tmp_rows[net]           //   8 bytes
            d = rows[net]               //   8 bytes
            term = term->nextterm       //  64 bytes
        end
    end
end
```

Figure 3.3: Case study: Sequence of objects touched by one of the main loops in twolf. Size of each object in comments.

nificantly better than just improving each of them in isolation. As we will show, irregular applications often exhibit different access patterns for each data structure in a single phase, combining spatial, pointer and indirect array-index access. This interleaving of different types of access is a challenge for prefetching methods that focus on just one type of access pattern [23, 44].

While 179.art and 183.equake have regular access patterns, the others interleave spatial and pointer access in complex ways. This interleaving may happen for three reasons. First, the application may perform strided access through an array while dereferencing pointer fields from each element (mcf:`nodes`, 188.ammp:`atoms`). Second, the application may perform strided access that uses the elements of one array to index into another (bzip2:`quadrant`, 300.twolf:`rows`). This is a form of pointer traversal that current pointer prefetching schemes [23, 76] often cannot detect. Finally, the application may access the elements of a data structure in irregular order, but each object may span multiple cache blocks that are accessed sequentially (ammp:`nodelist`, twolf:`netptr`) due to large object size or irregular object alignment in the cache. Such complex interleavings are a challenge to both spatial and pointer-based prefetch systems.

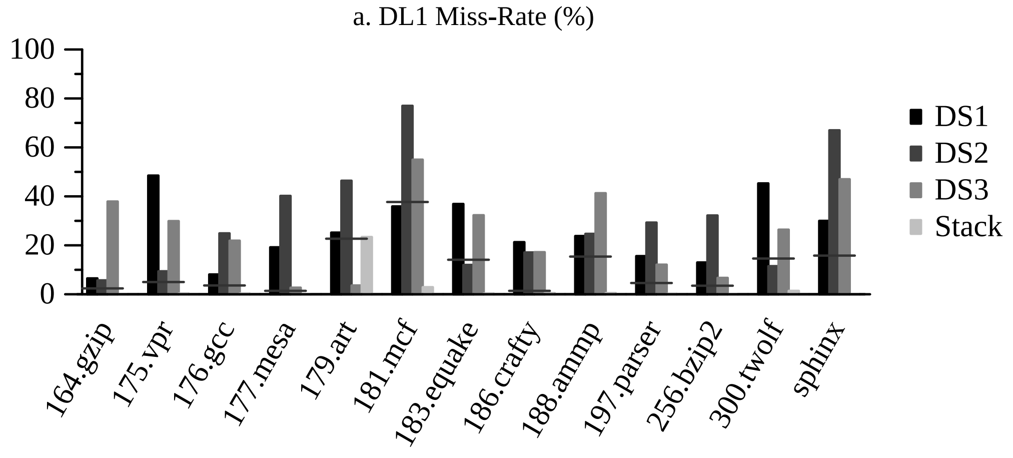
Access-pattern case study: We now perform a more detailed analysis to illustrate the potential for improvement from overlapping memory latency and the challenges in converting this potential. We focus on just one of our benchmarks - twolf - and look in its source code for insight into its behavior. Guided by the data profile in Figure 3.2 and by the more conventional code profile, our study yields Figure 3.3, the sequence of objects accessed in a crucial inner loop in twolf, responsible for 55% of all DL1 misses. This loop illustrates two interesting phenomena. First, while programs as a whole often have a large footprint, the footprint of each loop iteration in an irregular application fits easily in the DL1. Second, most misses in applications occur on the first access to an object in a loop iteration.

Since different data structures can access memory with a wide variety of access patterns in a single program phase, it is important for the system to optimize each according to its needs. Each loop iteration has a small footprint, so it is feasible to prefetch future iterations without disturbing the data for the current iteration. However, prefetching the data required for each iteration is challenging because it includes elements from different data structures with distinct access patterns. Taken together, these insights suggest a model where data streams into the processor in bundles of objects that each iteration will use. In the latter half of this dissertation we explore TwoStep, a concrete implementation of this model.

Having used the basic capabilities of DTrack to characterize our applications, we now explore novel uses of DTrack in asking and answering sophisticated questions on architecture design.

3.7 Case study: Data structure criticality

Our first case study concerns criticality of memory reference. Several recent studies have shown that not all cache misses are equally important as measured in the amount of latency that they expose to the processor [92]. In this context, does it make sense to simply use miss counts to select the data structures on which to focus our attentions? To answer this question we augment DTrack to detect cycles when no instructions are retired, and assign responsibility for each such *stall cycle* to the data structure referenced by the load or store at the head of the reorder buffer [91]. Our results show that for our applications the data structures that cause the most misses are



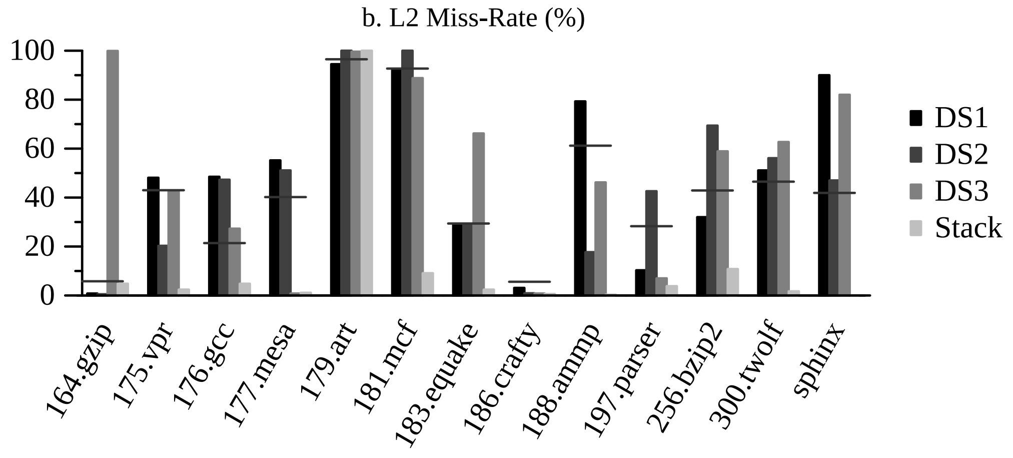


Figure 3.4: Decomposition of DL1 and L2 miss-rates by data structure. The aggregate miss-rate for each application is denoted by a horizontal line.

almost always also the ones responsible for the most stall cycles. There are two exceptions to this trend. The first is in 179.art; the array tds causes only 2.1% of all cache misses, but is responsible for 16.6% of all stall cycles. This data structure is critical because of the following loop that accumulates a subset of its elements:

```
for (tj=0;tj<numf2s;tj++) {
    if ((tj == winner)&&(Y[tj].y > 0))
        tsum += tds[ti][tj] * d;
}
```

This combination of data-dependent branches and computation serialized by tsum causes the infrequent cache misses in this loop to almost invariably stall the pipeline. Our conclusion is strengthened by a study of the source code. 179.art is a neural network simulator where learning occurs by iteratively modifying two arrays of top-down and bottom-up weights – tds and bus respectively. While these two arrays are largely accessed in very similar ways, the above loop is the only major access pattern not shared with bus. The second data structure that we observe causing a disproportionate number of stalls is the variable search in the chess-playing benchmark 186.crafty, which is responsible for 10.5% of all stall cycles in spite of causing just 0.2% of all cache misses. This global data structure contains the chess position being currently analyzed, and is used to display the board on screen. With the exception of these two data structures, the correlation between miss count and stall cycle count shows that data-structure criticality is of limited usefulness in the predominantly irregular programs that we study.

A related idealized experiment that provides indirect confirmation of this result explores the effect of selectively providing different data structures perfect single-cycle access to memory. To model this ideal behavior, we simulate cache misses to specific data structures in a single cycle, but continue to move data in these structures through the memory hierarchy so as to not give other data structures an unrealistically generous view of cache capacity. We find that selectively eliminating cache misses in even the most important data structure in an application has limited impact on performance in a majority of our applications. While there are a few exceptions, namely 188.ammp, 183.equake, it usually requires perfect memory for 2-5 major data structures to bring performance close to ideal. This result shows that future architectural and compiler enhancements will often need to optimize multiple data structures in different ways to significantly improve overall performance in memory-bound applications. It also shows that DTrack is indeed highlighting bottlenecks in the memory system when it ranks data structures by miss frequency.

3.8 Case study: Competition for caches

While Figures 3.2.a and 3.2.b show the distribution of accesses to the DL1 and L2, Figures 3.4.a and 3.4.b show the corresponding miss-rates at each level of the memory hierarchy. A common pattern in these figures is for

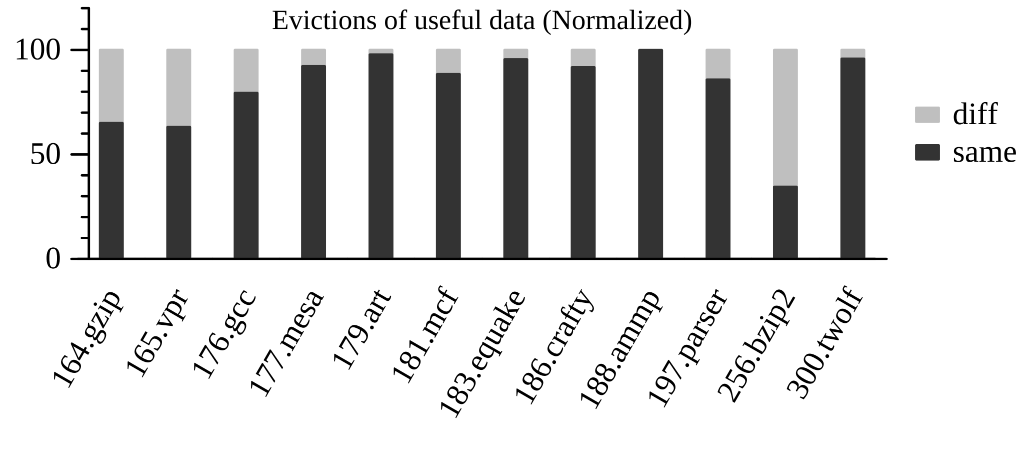


Figure 3.5: Breakdown of premature evictions. Useful data is only infrequently evicted by a different (diff) data structure.

a data structure with fewer cache misses to have a higher miss-rate. This pattern occurs as the major data structures compete with each other for limited cache capacity, so that a data structure that misses more often ends up with a larger fraction of the cache. While this is qualitatively a desirable response, such competition may cause suboptimal performance if different data structures repeatedly evict each other. If this behavior were found to be common, a computer architect may consider creating split caches [31] with static mapping policies assigning each data structure to a specific cache partition, or designing caches to bypass data in certain regions of a program's address space. Figure 3.5 shows how often useful data in the cache is prematurely evicted by a different data structure as opposed to the same one. With the exception of 256.bzip2 the majority of premature evictions are caused by conflict within a data structure, rendering a split cache by data structure unnecessary for these applications. This and the previous experiment are good examples of the ways that DTrack can help the computer architect with design decisions where traditional tools are unable to do so.

3.9 Summary

Analyzing our applications by data structure confirms and quantifies two nuggets of conventional wisdom that focus our attention in the rest of this dissertation:

1. *"Applications are not all alike."* The number of data structures that contribute 90% of an application's cache misses varies from 2 to 100. Applications with similar aggregate DL1 miss-rates of 20% can exhibit miss-rates of 2-40% for important data structures. The wide variety of behaviors, and the fact that not all applications have hot data structures, confirms the need for application-specific system adaptation.

2. Extremely irregular access patterns may be found in the wild. 181.mcf performs bounded depth-first-search over sub-trees; 300.twolf and 256.bzip2 perform lots of indirect array access; 188.ammp interleaves random pointer traversals with spatial access over each 2KB object. As a result, cache misses largely occur on the first access to an object in a loop iteration, and predicting the object each iteration will access can be difficult.

The combination of these insights leads us to a prefetch system biased towards complex access patterns. Since the footprint of any given loop iteration is tiny relative to cache capacity, we focus on orchestration at the loop iteration granularity.

In addition to these insights, DTrack influences the rest of this dissertation in two methodological ways. First, it provides valuable infrastructure for debugging optimizations as we describe later. Second, our analysis of critical loads in Section 3.7 suggests a metric to evaluate optimizations in scheduling - reduction in stall cycles. Scheduling does not eliminate cache misses for irregular programs without much spatial locality. Thus, cache miss counts and rates should remain unchanged in the presence of prefetching. Measuring reduction in stall cycles provides a solution to this problem, quantifying the latency tolerance of a prefetching approach. One additional wrinkle is that critical paths can be easily shifted by improvements or changes to the application [28, 92]. This suggests refining our metric to stall cycle reduction by data structure, which gives us a richer picture of how well a technique addresses the perceived problem, and also of how much speedup we obtain before hitting the next bottleneck.

In the next chapter, we extend these insights to phase behavior, again using novel methodology to quantify phase variation in access patterns, and providing key infrastructure for selecting good simulation intervals from a high-level perspective. Our characterization then drives the design of TwoStep, which provides a parsimonious basis set of mechanisms to give each major loop in an application a carefully tuned prefetching strategy, specifying what to prefetch, when to prefetch it, and where to prefetch it to.

Chapter 4

Phase analysis

This chapter extends our high-level characterization of applications by decomposing application behavior by data structure *and* global program phase, and by translating this decomposition into a summary of major application access patterns that is used in the design of TwoStep in the next chapter. In the process, we make two contributions to the state of the art in phase analysis methodology.

Phase behavior has received much attention in recent times [6, 52, 68, 82], with the eventual goal of designing system hardware/software to adapt to changing application requirements. Studies using Basic Block Vectors (BBVs) explore their uses in clustering phases and detecting phase transitions in an offline [83, 84] and online [86] setting. One alternative to BBVs is the use of program counter or Extended Instruction Pointer Vectors (EIPVs) [6]. Another alternative consists of more high-level metrics based on code structure, such as register use vectors or loop vectors [52]. All these studies share a common workflow. They generate the variation of some metric (such as cache miss-rate) over time, aggregated in some *sampling period*. They then try to identify regions with 'similar' behavior and the boundaries between such regions. All these studies suffer from two major drawbacks:

- They operate on aggregate phase data for detecting phase behavior. While this is sufficient for some applications, we show in Section 4.1 that it can hide details of memory behavior.
- They select their sampling period in an ad hoc fashion and use a single sampling period across all their applications to automatically detect phase boundaries [82]. Nagpurkar et al. recently showed that the notion of phase boundaries is not absolute, and that the phase boundaries one picks and the granularity at which to view them depend on their eventual purpose [68]. This result suggests that automatic phase-detection algorithms are deeply influenced by the sampling period at which data is provided to them.

Our methodology addresses both drawbacks. In Section 4.1 we use DTrack to measure phase behavior on a data structure basis. In Sections 4.2–4.4 we demonstrate a new technique based on spectral analysis that automates the process of selecting a good sampling period for phase data. Rather than pick an ad hoc sampling period and then automatically determine phase boundaries at that granularity, we automate sampling period selection to yield a phase graph where global phase behavior is more readily apparent.

Applying these two methodological improvements, we quantify the phase behavior for each application at an application-specific sampling period in Sec-

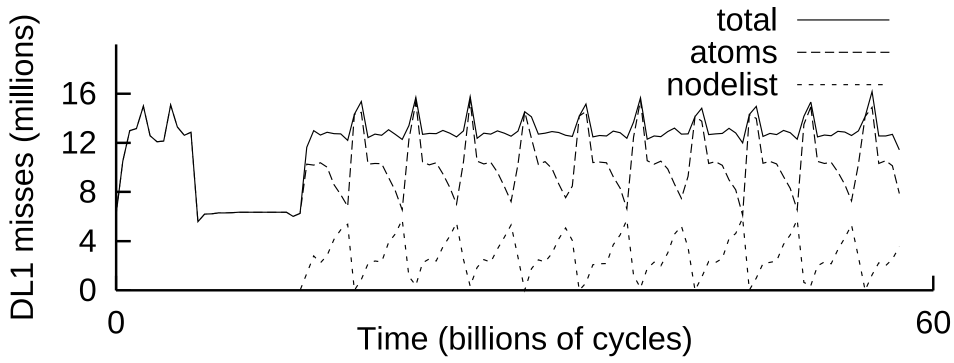


Figure 4.1: Just tracking total misses can miss interesting effects. DL1 cache misses in aggregate and by data structure in `188.ammp`.

tion 4.7. Our results show that data structures have very different access patterns in different phases; however all data structures in an application largely share the same phase boundaries. We use this phase data in Section 4.8 to determine the dominant access patterns in each application, a high-level insight that is used to drive the design of the TwoStep prefetching system in the rest of the dissertation.

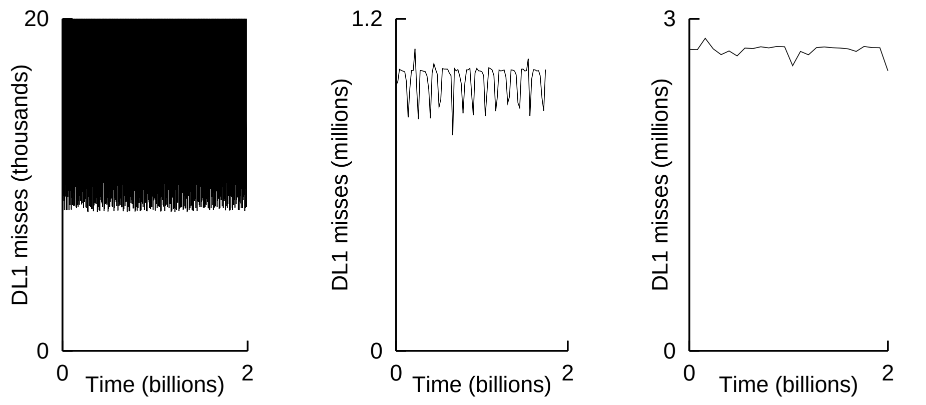
4.1 Analyzing phase behavior by data structure

Studying phase behavior by data structure is important; looking at the time-varying behavior of aggregate misses alone can be misleading and hide important data structure interactions. Figure 4.1 illustrates this: the data structures `atoms` and `nodelist` in `188.ammp` are consistently anti-correlated. As one increases the other decreases and vice versa. Studying just the curve for total cache misses would miss this interaction and also underestimate the degree to which the application's behavior is changing under the surface. The reduced amplitude changes also make automatic phase detection more difficult, as we explore later in this chapter.

This pattern is not uncommon; six of the nine applications we study exhibit significant differences in data structure miss distribution in different phases. Therefore in the rest of the results in this chapter we use our DTrack toolchain to generate time-varying miss-count or miss-rate data for individual data structures rather than for the aggregate application as a whole.

4.2 Sampling period selection: Overview

Our second methodological innovation is a technique to view time-varying behavior at a sampling period that highlights global phase transitions. Our technique is based on two insights from spectral analysis: that increasing sampling period is a process of aggregation that has a damping effect, and that global phase behavior consists of emphasizing rare (low-frequency) transitions over common (high-frequency) ones. Figure 4.2 shows the temporal variation in DL1 miss count for a single data structure in 183.equake by aggregating miss count at three different sampling periods: one sample every 10 million cycles, one sample every 180 million cycles, and one sample every 500 million cycles. This figure illustrates a general trade-off for phase analysis, either offline or online. Offline, overly frequent sampling puts too many data points on a graph, making global trends harder to detect. Online, frequent sampling increases overheads. Conversely, increasing sampling period too much reduces the information content to close to that of aggregate DL1 misses, defeating the



a. DL1 misses every 10 million cycles *b*. DL1 misses every 180 million cycles *c*. DL1 misses every 500 million cycles

Figure 4.2: *Selecting a sampling period, step 1:* Phase behavior curves corresponding to a stream (183.equake) at different sampling periods. The challenge is to select a sampling period that is neither too noisy (*a*) nor over-damped (*c*), but just right (*b*).

purpose of phase analysis, whether offline or online. We would like to avoid both classes of degenerate data collection.

We begin our description of this process by outlining the various stages involved in our offline methodology, and by introducing some terminology in the process. First, we generate a *stream* of data-structure-specific data using DTrack at a low sampling period of one million cycles. To model larger sampling periods we aggregate the points in this stream to generate various *curves* such as the ones shown in Figure 4.2. We then use a simple *volatility* metric — described in the next section — to compute the volatility of these curves, and combine the volatilities at different sampling periods to generate a *volatility profile* for the stream. This process is graphically depicted in Figures 4.2–4.8.

Volatility profiles provide a concise summary of the phase behavior of an application at different granularities; we show that they suggest good sampling periods in a straightforward manner: low granularities with low volatilities. We now describe each stage in detail, dwelling on the intuitions behind our design decisions and the alternatives we considered.

4.3 Sampling period selection: The volatility metric

Phase boundaries in a curve are dramatic changes in amplitude over time. In selecting the right granularity to detect phase boundaries we would like to highlight only the most important such dramatic changes. Thus, the volatility of a curve should answer the question: what is the largest magnitude of amplitude change commonly seen in the curve? Let us begin by answering this question for the degenerate case: with a curve containing just two points. We denote the curve consisting of the values X_1, X_2 in adjacent time steps as $[X_1, X_2]$.

Volatility at a point: The curve $[1.1, 1.2]$ has much lower volatility than the curve $[1, 10]$. This intuition is adequately captured by our conventional notion of relative change, or growth. A variable that doubles between adjacent sampling intervals demonstrates higher volatility than one that grows or shrinks by 10%. We formalize this notion into the following volatility metric at a given time step. Given a stream $[X_1, X_2, X_3 \ldots]$, the volatility at each time step is defined as:

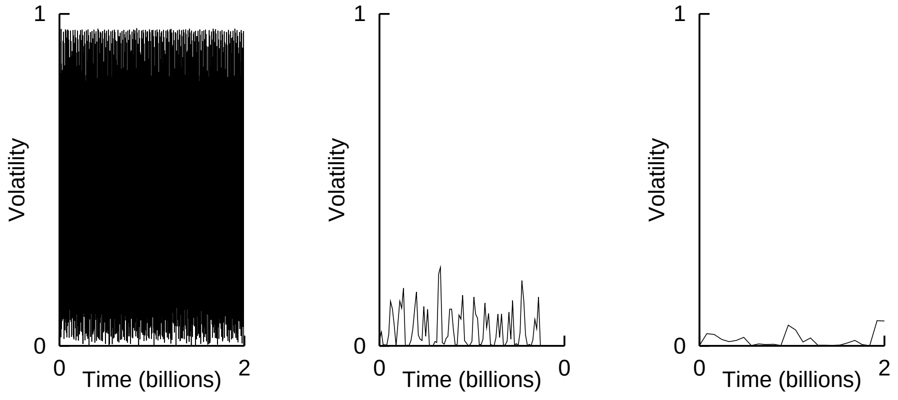
Curve	Point volatilities
$[1, 1, 1, 1, 1]$	$\{0, 0, 0, 0\}$
$[1, 1, 1, 1, 2]$	$\{0, 0, 0, 0.5\}$
$[1, 2, 1, 2, 1]$	$\{0.5, 0.5, 0.5, 0.5\}$
$[1, 10, 1, 10, 1]$	$\{0.9, 0.9, 0.9, 0.9\}$

Table 4.1: Computing the point volatilities of some simple example curves.

$$g_t = \frac{abs(X_t - X_{t-1})}{max(X_t, X_{t-1})} \tag{4.1}$$

g_t is similar to the conventional notion of 'growth', except that it is symmetric: g_t is 0.5 whether X_t has doubled ("grown by 100%") or halved ("shrunk by 50%") since the last time step. This symmetry ensures that the volatility between two values is the same regardless of which comes first. By this definition, the curve $[1, 10]$ has a volatility of 0.9, while the curve $[1, 1.2]$ has a volatility of 0.1. Even more trivially, the curve $[1, 1]$ has a volatility of 0.

Summarizing the volatility of a curve: Given the above formulation for the volatility of a 2-point curve, we can now view a curve with n points $[X_1, X_2, X_3 \ldots]$ as a set of 2-point curves $\{[X_1, X_2], [X_2, X_3] \ldots\}$, and we can now compute the *point volatility* for each of these. Table 4.1 shows the point volatilities of some simple example curves. Notice that each point volatility lies in the open interval $(0, 1)$, that equal adjacent values yield a point volatility of 0, and that rapid increases and decreases in value cause high volatilities. Figure 4.3 illustrates this process for a curve with more points, showing the



a. 10 million cycles *b*. 180 million cycles *c*. 500 million cycles

Figure 4.3: *Selecting a sampling period, step 2:* Corresponding point volatilities for each point in the graphs of Figure 4.2.

corresponding point volatilities g_t for the curves in Figure 4.2.

Summarizing the volatility of these 2-point curves is an exercise in statistics, and there are many candidate ways to do so, starting with simple ones such as mean, median and mode. To select a good method of summarization, recall that the goal is to determine the largest volatility that is commonly seen in the curve. This requirement can be broken down into two sub-requirements: first, that all commonly occurring volatilities be considered in our aggregation; and second, that rare volatilities not be considered. Figure 4.4 provides an alternative way to formulate our requirement: curves with similar magnitudes of high-frequency noise must have the same volatility, regardless of their low-frequency phase behavior. Let us consider the three simplest alternatives for aggregating point volatilities in the light of these requirements:

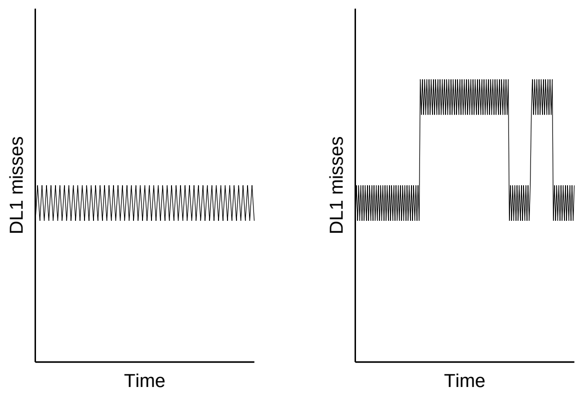


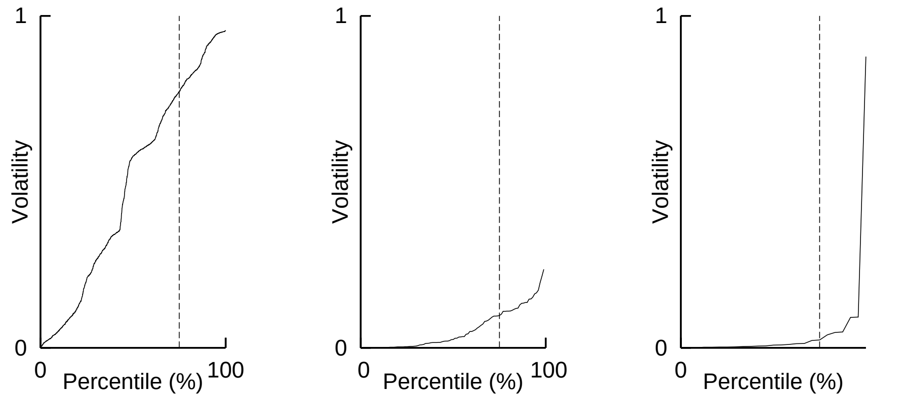
Figure 4.4: Property of a good volatility metric: both these curves should have the same volatility, as an indication of how much noise is added by the common transitions, while ignoring the rare transitions.

- *Mean.* The mean of a set of values is sensitive to infrequent outliers. This violates our second constraint. It can also cause a set of high point volatilities to be 'smeared down' into a lower value in the aggregate. For instance, the set of point volatilities $\{0, 0, 0.5, 0.5\}$ translates to an average of 0.25 which underestimates the common volatility of 0.5, violating our first constraint.

- *Median.* Consider the set of point volatilities $\{0, 0, 0.25, 0.5, 0.5\}$. The median 0.25 violates our first constraint: 0.5 is larger and common.

- *Mode.* The median is completely unrelated to our requirements and dramatically incorrect for sets with balanced frequencies: $\{0, 0, 0, 0.9, 0.9, 0.5\}$ would yield 0, violating both of our constraints.

Thus, none of these are suitable. However, this quick thought experiment yields one major insight: that we need to fix precisely what we mean by 'com-



a. 10 million cycles *b*. 180 million cycles *c*. 500 million cycles

Figure 4.5: Selecting a sampling period, step 3: Sort the point volatilities for each graph in Figure 4.3. The volatility of the curve is defined as the point volatility at the 90th percentile.

mon' or high-frequency. While mean and mode entirely fail to capture our requirements, the problem with using the median is relatively minor, and is corrected by increasing the percentile at which to place the maximum bound. Thus, selecting the median or 50th percentile could miss a higher point volatility that occurs nearly 50% of the time, selecting the 70th percentile could miss a higher point volatility that occurs at most 30% of the time, and so on. We empirically find that selecting the 90th percentile, which excludes 10% of the largest point volatilities, gives us a good measure of the largest and commonest transitions in a curve — the high-frequency noise. Thus, we generate the volatility of a curve from the set of its point volatilities by sorting the point volatilities in ascending order and reading off the point volatility occurring at the 90th percentile. Figure 4.5 illustrates this process for our running example curves of Figure 4.3. In each graph, the dotted line denotes the 90th

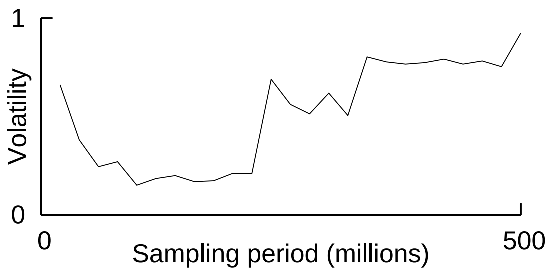


Figure 4.6: The volatility profile for the data structure `inbuf` in `164.gzip`, showing volatilities for curves aggregating from 1 to 500 million cycles worth of DL1 cache misses together.

percentile and the point volatility at this percentile is treated as the volatility of the entire curve.

4.4 Sampling period selection: Volatility profiles

We have thus far determined a suitable volatility metric quantifying the amount of high-frequency noise in a curve. The next step is to use this volatility metric to determine a suitable sampling period for a given stream. To do so, we first compute the set of curves corresponding to the input stream when aggregated at different sampling periods. For each curve we determine the volatility as described above. Plotting the volatility of the curve against the sampling period at which it was gathered yields the *volatility profile* for the underlying stream.

Figure 4.6 shows one such volatility profile. Putting together our methodology in every stage so far, this graph is generated as follows. We configure DTrack to emit miss-count statistics by data structure every 1 million cycles and run 164.gzip on top of it. This experiment yields us a stream of data points corresponding to the DL1 miss count for inbuf in every million-cycle interval of execution. Aggregating these data points in different ways yields curves for the DL1 miss counts every 2 million cycles, every 3 million cycles, and so on. We compute the volatility for every such curve from sampling period of 1 million to 500 million cycles, and plot the resultant volatilities against sampling period to yield the graph of Figure 4.6. The points on this graph with relatively low volatilities represent sampling periods where global phase behavior is more salient and easily discerned. The next two sections now elaborate on the process of selecting a good sampling period given the different types of volatility profiles.

4.5 Results: Volatility profiles

To generate volatility profiles for our applications, we apply the procedure from the previous section on streams for DL1 and L2 miss count and miss-rate of the most frequently missing data structures as generated by the methodology outlined in Section 3.3. Across the applications we study, we find that the DL1 and L2 miss counts for different data structures largely exhibit volatility profiles with the same trends, and with minima at the same sampling periods. Therefore, we focus on the DL1 miss-count stream for a single major data structure in each of our applications. The left-hand graphs in Figures 4.7 and 4.8 summarize the volatility profiles for these data structures.

The volatility profiles in Figures 4.7 and 4.8 may be classified into three categories. First, 175.vpr, 179.art, 181.mcf, and 300.twolf show consistently low profiles, so that an arbitrary selection is likely to highlight global phase behavior. Second, 177.mesa, 183.equake, and 256.bzip2 exhibit monotonically decreasing volatility profiles as a result of the natural damping effects of aggregation with increasing sampling period. In these cases we empirically select the smallest sampling period with a volatility of less than 0.2. The third and final category consists of 164.gzip and 188.ammp, applications where the volatility profile is more complex. We explain these volatility profiles in greater detail in the next section, and describe our more ad hoc methodology to determine good sampling periods for these applications.

4.6 Explaining and handling non-monotonic volatility profiles

The variety of volatility profiles in Figures 4.7 and 4.8 bears some scrutiny. We began this chapter with the assumption that the damping effect of aggregation would cause volatility to monotonically drop with increasing sampling period. However, our results show that this is not always the case; 164.gzip and 188.ammp have particularly complex, non-monotonic volatility profiles. These phenomena are explained by the discrete set of sampling periods available to us, and the interaction of these discrete points with the intrinsic periodicity of an application.

At a high level an application consists of nests of loops that access different data structures in different ways. The access pattern of a given data

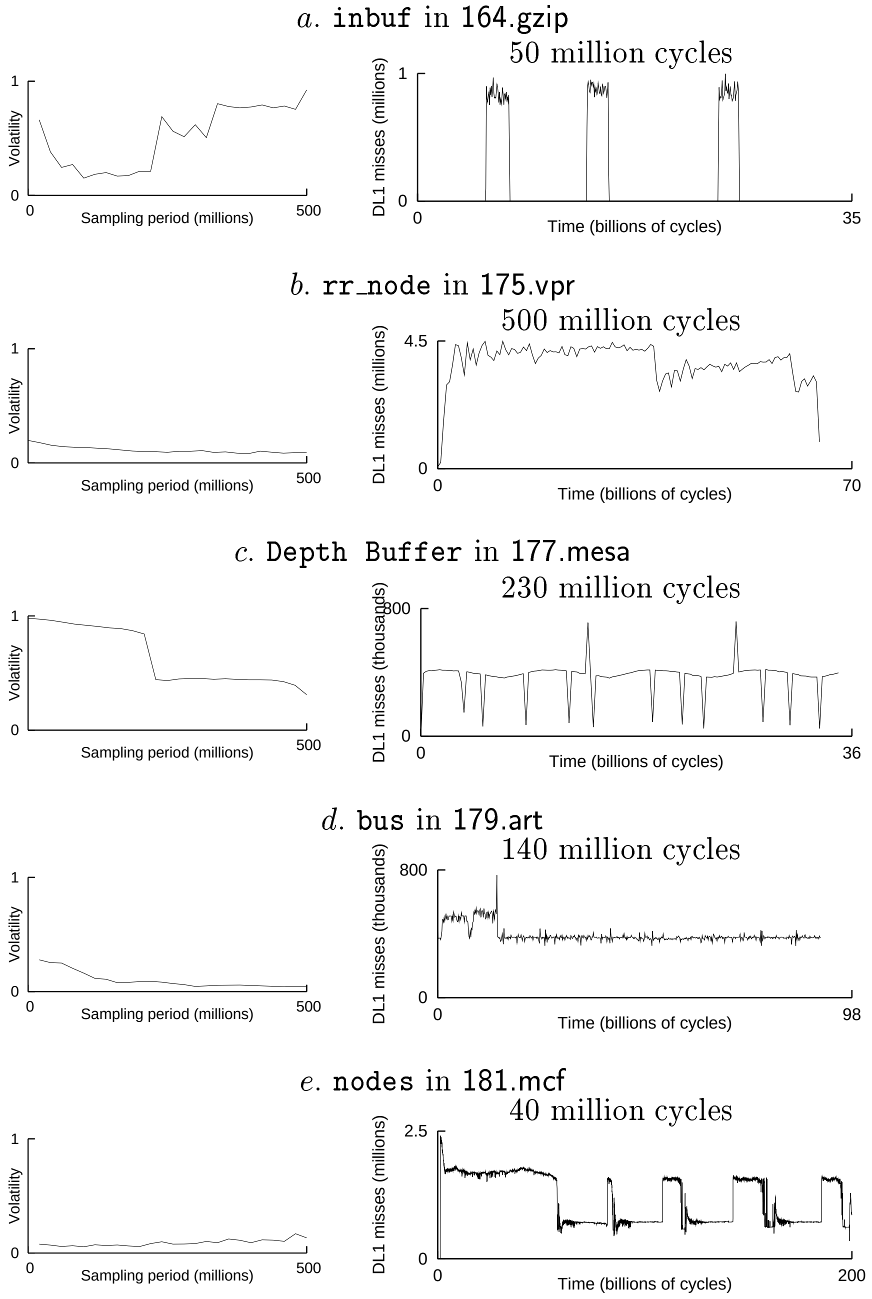
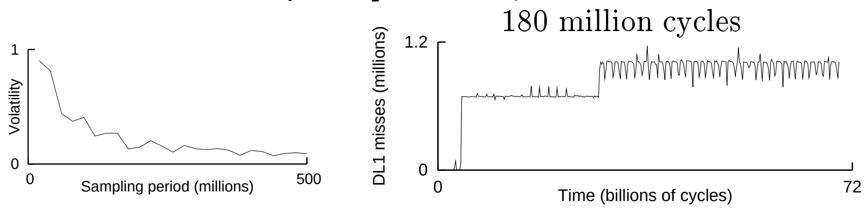
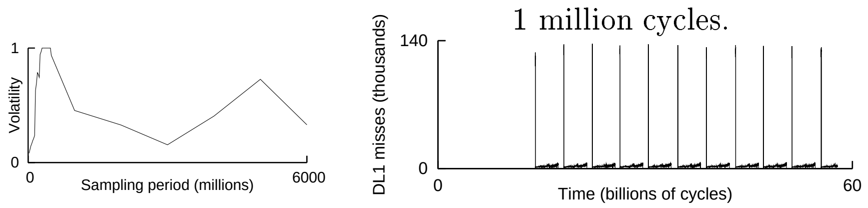


Figure 4.7: Volatility profiles of some major data structures in our applications (left), and the corresponding phase behavior (right) at one low-volatility sampling period in the profile (specified above each right-hand graph).

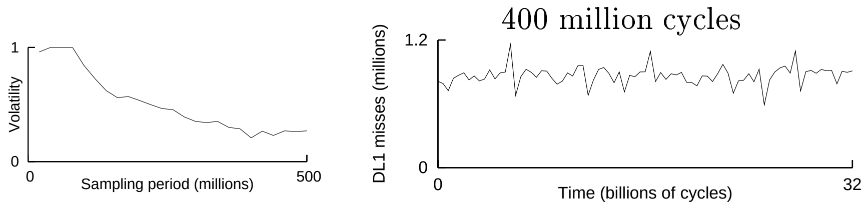
f. disp in 183.equake



g. atomlist in 188.ammp



h. zptr in 256.bzip2



i. tmp_rows in 300.twolf

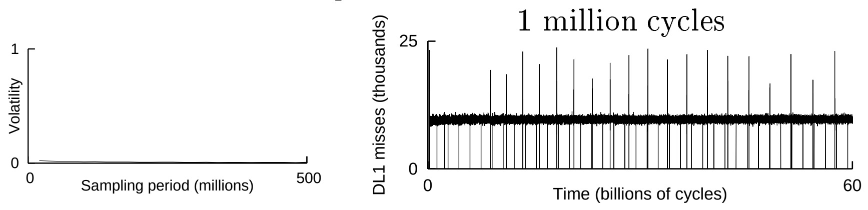


Figure 4.8: Volatility profiles of some major data structures in our applications (left), and the corresponding phase behavior (right) at one low-volatility sampling period in the profile (specified above each right-hand graph).

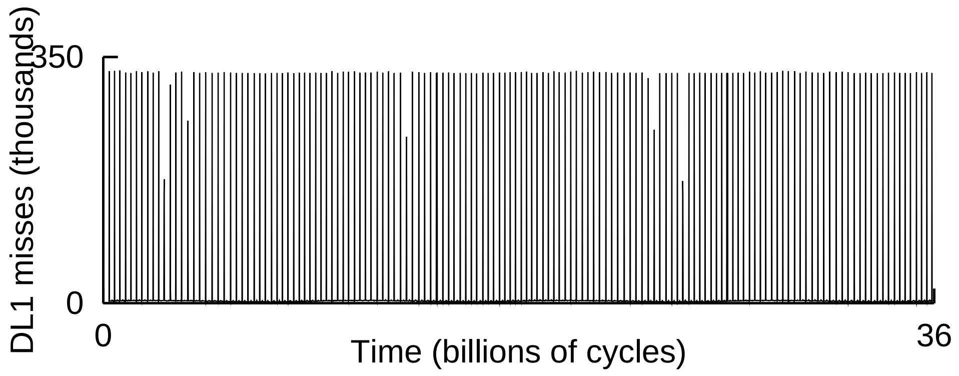


Figure 4.9: The phase behavior of 177.mesa at 10 million cycles. Compare with Figure 4.7*c*.

structure in a given loop may contribute a component with a certain approximate period to the phase behavior of the data structure. Combining all the interacting periodic components corresponding to a data structure yields the overall phase behavior of that data structure. If all the components for a data structure have relatively low time periods and high frequencies, we expect aggregation at high sampling periods to smooth out their disparate periodic effects. If a stream contains a component with a substantial time period, however, we observe a steeply oscillating volatility profile, with troughs at factors and multiples of that time period.

Such streams with coarse-grained periods make it more difficult to select a sampling period, requiring volatility measurements at a large number of values in order to find good candidates. For example, if a stream is dominated by a period of 7 million cycles, taking measurements at sampling period increments of 10-million could fail to identify a good sampling period. By the time we find low volatility (at a sampling period of 70 million cycles) we may have damped out all phase behavior. Understanding such interactions in application phase behavior is a challenge for future research. In the context of this study, finding a low-volatility sampling period required gradually refining volatility measurements for 177.mesa and 188.ammp. As a concrete example of this, Figure 4.9 shows the phase behavior seen for `Depth Buffer` in 177.mesa at a sampling rate of 10 million cycles. Comparing this curve with that in Figure 4.7*c* shows how widely dissimilar different a stream can look at different sampling periods, and how selecting a bad sampling period can occlude gradual periodic patterns. The global phase behavior seen in Figure 4.7*c* is only observable in a narrow window of sampling periods, from 200 to 300 million cycles. Offline phase detection techniques that fail to use sampling periods in this range would show either too many phase transitions or too few, occluding the more gradual phase behavior in either case. Similarly, online phase detection techniques that fail to adjust the sampling period would be unable to adapt effectively to the changing requirements of this application.

Summary: The goal of the last 4 sections has been to come up with a rigorous methodology to select a good sampling period at which to view and operate upon graphs of temporal behavior. Our proposed methodology, based on a volatility metric, fulfills this purpose by concisely summarizing the merit of every point in the state space of possible sampling periods. The next step, of selecting a good sampling period, is more ad hoc. The lack of full automation is a result of one major factor: efficiency considerations force us to maintain a lower bound on the granularity at which we can vary sampling period. Interactions between this sampling period and intrinsic periodicities of different streams force us to manually inspect phase graphs for some applications at a few low-volatility sampling periods before settling on the period with the cleanest expression of global phase behavior. Our general heuristic, though, is to select the lowest possible sampling period with a low enough volatility. This corresponds to points to the bottom and left in our volatility profiles.

4.7 Results: Phase behavior at a good sampling period

Having described in detail the procedure for selecting a good sampling period for each of our applications, we can now study the phase behavior of each application at this application-specific sampling period. The right-hand side graphs in Figure 4.7 and 4.8 summarize the phase behavior of the DL1 miss count for one major data structure in each of our applications. Each of these graphs is labelled with its sampling period of N cycles as selected from the volatility profile on the left, and plots DL1 miss-count for a single data structure per N cycles.

Our results can be broken down into three categories. First, applications with no phase behavior past initialization: 179.art, 183.equake, and 300.twolf. Second, those with simple phase behavior between a well defined set of phases with easily-discerned boundaries: 164.gzip, 181.mcf and 188.ammp. Third, more complex curves with poorly defined phases and fuzzy phase boundaries: 175.vpr, 177.mesa and 256.bzip2.

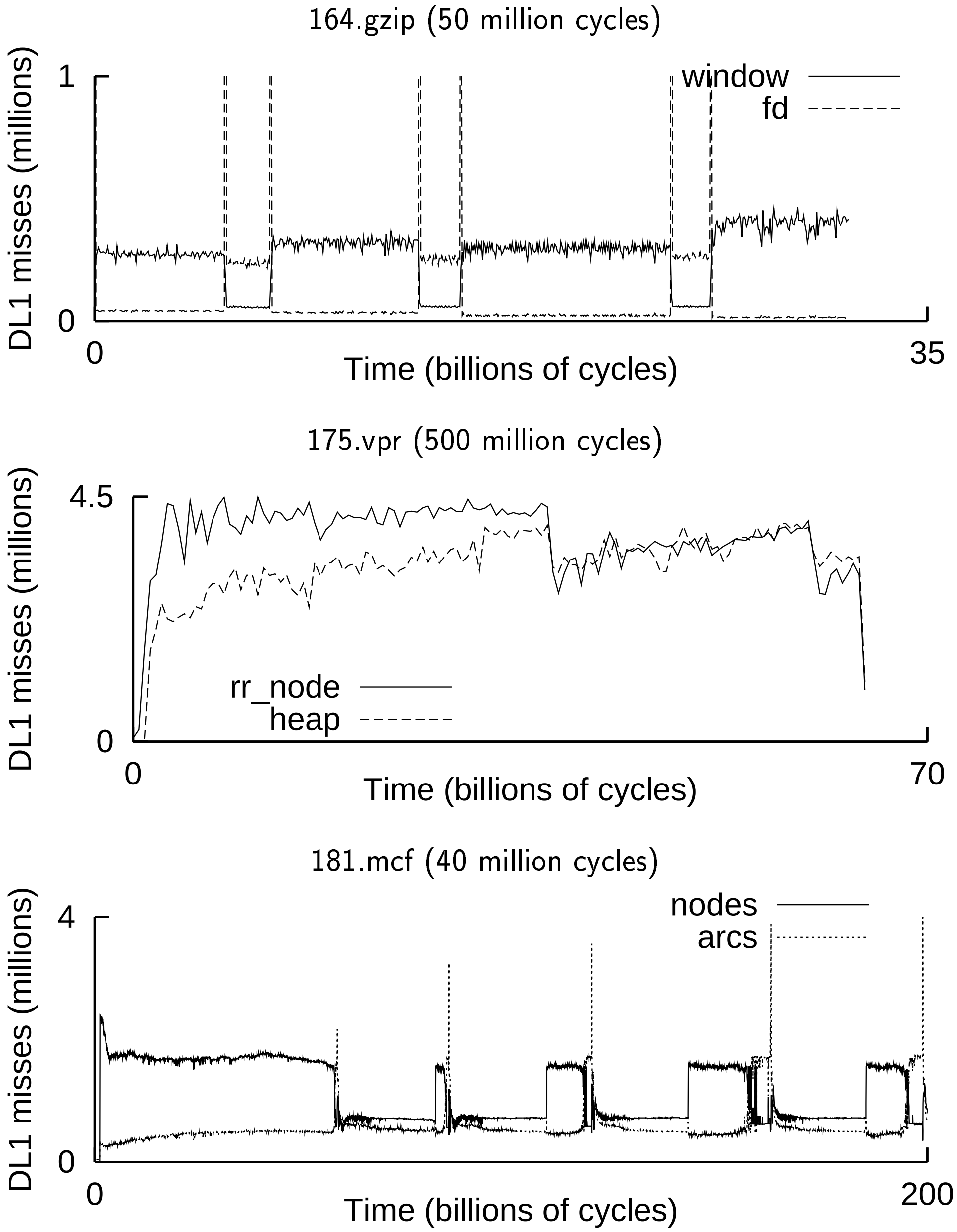


Figure 4.10: Applications with inversion: a different data structure contributes the most misses in each phase. (sampling period in parentheses)

Categories 2 and 3 both contain applications with phase inversions, where a different data structure contributes the most cache misses in each phase. Figure 4.10 shows the phase behavior of the major data structures in those of our applications with such inversions – 164.gzip, 175.vpr and 181.mcf. We use this data on phase transitions and inversions in these applications to distill each of our applications down to a concise description of their major access patterns.

4.8 Results: Translating phase behavior into access patterns

The phase behavior of an application can be used for a variety of purposes as detailed in the next section. In this disseration we use it to help drive the design of the TwoStep prefetch system in the second half of this dissertaion. Combining our insights from DTrack with code profiles allows us to identify the different access patterns in each phase, and the roles of different data structures where inversions occur. By manually correlating code profiles, the data profiles generated by DTrack, and the phase behavior data from the previous section, we are able to concisely summarize the major access patterns in each of our applications.

- 164.gzip consists of alternating phases that read a section of input data into a buffer, and compress the contents of the buffer. Both phases have sequential access patterns with lots of spatial locality.

- 175.vpr consists of two data structures: a `heap` of objects, each containing a `rr_node`. The heap is accessed in a halving or doubling stride, while rr_nodes are more irregular. The interleaving of accesses to the two is highly data driven.
- 179.art consists of two 2-D arrays: `bus` and `tds`. Both are accessed simultaneously and sequentially.
- 181.mcf consists of alternating phases of depth-first-search over a subtree of `nodes`, and heap sort over a heap of `arcs`.
- 183.equake consists of regular sequential access over several 3-D arrays.
- 188.ammp consists of a linked list traversal through `atomlist`, interspersed with a pass of much more irregular access every 12-15 iterations in order to update 200 pointers to spatially neighboring atoms.
- 256.bzip2 performs irregular indirect array accesses over three distinct arrays — `zptr`, `block`, and `quadrant` — using indices in one array to access another.
- 300.twolf contains a single phase with a complex access pattern summarized earlier in Figure 1.2, interleaving spatial, pointer and indirect array access.

These access patterns drive the design of several aspects of the TwoStep prefetch system in the second half of this dissertation. These aspects include the basic insight that such a wide variety of techniques requires compiler-driven policies to determine what to prefetch, the design of the ISA for the TwoStep prefetch controller, a quantitative analysis of the timing constraints on dependent prefetches to determine that the controller must be placed at the L2, and the need for auxiliary structures and efficient flow control in order to perform prefetching into the DL1. We explain these considerations in more detail in the next chapter.

4.9 Summary

As computers have become cheaper and more accessible the trend in the last 30 years has been for applications to grow more diverse (with new categories like streaming media and personal productivity), more complex (word processors check grammar and also perform speech recognition and synthesis) and more memory-intensive. These trends are likely to continue in future: the number of applications running concurrently on a system, the variety of applications, and the variety of phase behaviors in an application are all likely to increase. In the face of these trends, one-size-fits-all heuristics are insufficient, and adaptive approaches increase in importance.

Our response to these trends has been a detailed characterization of nine applications with a wide variety of access patterns, first decomposing their aggregate memory hierarchy behavior by data structure in the previous chapter, and then further decomposing these results by global program-execution phase. Our detailed characterization yields a concise summary of the major access patterns that we use to drive the design of TwoStep in the rest of this dissertation.

While we focus on a single application for this detailed characterization, our novel methodology methodology can be applied to systems research in a variety of ways. In the past, identifying phase behavior has been useful in several areas, such as adaptively varying processor issue width or cache capacity [6, 86]. Our data shows that augmenting these past online approaches with ways to adaptively tune the granularity of phase transition decisions will increase their effectiveness. Tuning phase granularity online is an open problem that will need to be addressed in future. In offline phase analysis, combining prior implementations with data structure decomposition and the correct sampling period can provide a more rigorous framework for phase analysis and more sophisticated insight into many areas of application behavior.

Chapter 5

TwoStep: Precomputation-based prefetching with lightweight throttling

Our study of data structures and phase behavior in different applications shows the wide variety of access patterns modern systems have to deal with. The second half of this dissertation describes and evaluates our approach to *application-driven prefetching*, a precise set of mechanisms that allow individual applications to be optimized at runtime according to their needs and access patterns. Our prefetch system is called TwoStep. TwoStep combines compiler-generated precomputation threads, a prefetch controller in the L2 that runs ahead of the main program, and lightweight mechanisms for flow-control and throttling. It is designed to work in the presence of truly complex access patterns interleaving pointer and spatial access that prior approaches have struggled with. In the rest of this chapter, we describe the challenges presented by such applications to previous approaches, describe the design decisions that led to TwoStep, and provide initial results over a set of hand-crafted kernels for four of our applications in order to show the soundness of the basic microarchitecture design.

5.1 Drawbacks in past approaches

A variety of mechanisms have been used in prior prefetching studies. We now survey the prior work on prefetching in terms of its constituent mechanisms separated along four directions: where a prefetch originates, what to prefetch, when to prefetch it, and where to prefetch to. The process of the survey recapitulates the rationales for our design decisions for TwoStep.

Prefetch origin: There are three broad choices in deciding where prefetches should originate: in the main processor as part of the application program [12, 41, 56, 59], in the main processor as a separate thread [11, 66, 93], or in the lowest level of the cache hierarchy facing main memory [47]. While the latter requires more overhead and book-keeping to orchestrate, it has an advantage that DTrack tells us is crucial: it reduces the latency between dependent prefetches. Since prefetches have to go only one way from L2 to processor, both baseline latency and queuing delay due to bandwidth constraints are minimized. The cost is additional hardware complexity.

What to prefetch: There are 4 broad choices in deciding what to prefetch: addresses spatially close to recent addresses [12, 55, 59], recently-fetched cache-lines for pointers [23], pattern detection tables (stride or address-correlation) in hardware [40, 41], and finally compiler-generated addresses [56, 59]. Of these, the first three are tuned to narrow varieties of access-patterns; responding to arbitrarily complex access patterns requires compiler intervention. The cost is compiler complexity. Also, compiler-based prefetch schemes in the past have often struggled with the next decision of prefetch timing.

When to prefetch: There are two opposing constraints on timing prefetches: prefetches need to occur early enough relative to use to overlap their entire latency. They also need to occur close enough to the use not to evict more proximally-useful data and cause *cache pollution*. Past approaches on timing prefetches have largely been constrained by the design decision of what to prefetch: compiler-based approaches [56, 59] have relied on the compiler to time prefetches as well, resulting in brittle strategies that cannot adapt to changing runtime requirements; hardware-based approaches [40, 41], have struggled to issue prefetches early enough since the microarchitecture's view is more local than a compiler's. There has been recent work on issuing systems of prefetches [55], often under compiler guidance [97, 99] rather than single prefetches at a time in order to increase available slack. This approach is the most promising among the alternatives. However, the challenge is to meet conflicting timing constraints without running into either the drawbacks of software approaches (rigid strategies) or hardware ones (overhead in detecting and avoiding pollution). The prioritization decision between independent sequences of prefetches [22, 102] can also cause design complexity.

Where to prefetch to: This decision presents 3 major options: prefetch to the L2, prefetch to the L1 or prefetch to an auxiliary structure connected to the caches. Prefetching to the L1 is a challenge because its small capacities increase the risk of pollution. As a result, most recent approaches have prefetched only to L2. Spatial prefetch schemes have explored prefetching to an auxiliary structure called a stream buffer [42, 85] in order to avoid L1 pollution, but at the cost of a slightly increased latency somewhere along the critical path of cache accesses. Stream buffers impose ordering constraints on the use of prefetches, however; as a result they have not been used with success for irregular applications.

This analysis highlights the issues in prefetching for highly irregular access patterns. We would like to have the compiler select what to prefetch but decouple the decision from prefetch timing. We would like to issue prefetches far in advance from the L2 but allow the processor to control the prefetch thread to avoid pollution. We would like to prefetch to L1 but avoid pollution. Our key insight is that decoupling each prefetch into 2 stages solves all these problems with low cost in design complexity or overhead. We now describe our aptly-named TwoStep prefetch scheme.

5.2 An overview of TwoStep

Figure 5.1 shows a high-level schematic for our TwoStep microarchitecture, highlighting the major components of the prefetch system - the prefetch controller in the L2, the FIFO between L2 and DL1, and ISA enhancements to orchestrate data transfer between FIFO and DL1. TwoStep performs long-range prefetching in the L2 under the direction of a compiler-generated *prefetch*

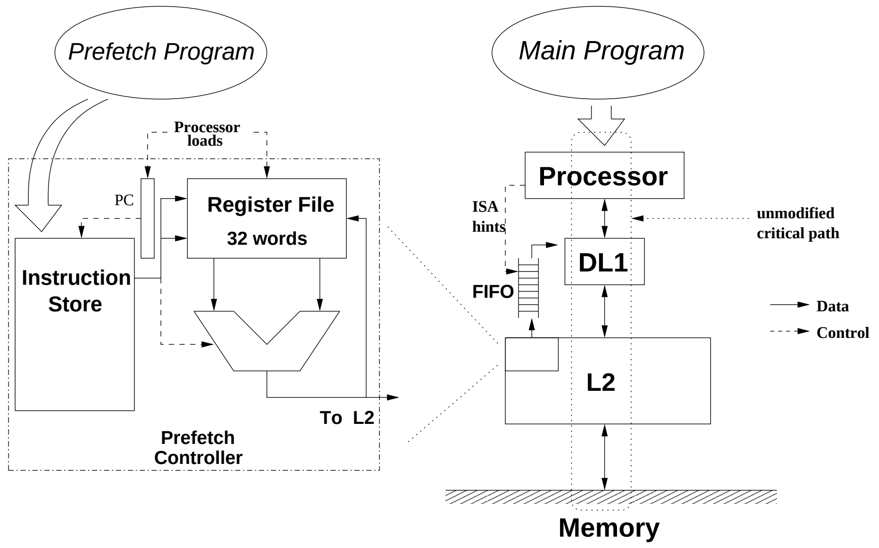


Figure 5.1: The TwoStep prefetching system

program, and short-range prefetching to orchestrate the transfer of data into the DL1 without polluting it. The L2 prefetch controller is a simple single-issue in-order processor. A prefetch program is loaded into the prefetch controller when its corresponding main program is loaded into the processor. Prefetching is triggered when the main program reaches specific program phases. At the start of a program phase for which the compiler decided to enable prefetching, compiler-inserted code in the main program initializes various registers in the prefetch controller, including the prefetch PC, and signals the controller to begin prefetching. At this point the prefetch controller begins executing its loaded program. Load instruction types in the prefetch program (the most frequent category) cause the object (with a statically well-defined size) in the result register to be prefetched. When such an address is not available in the L2, it is requested from main memory and the prefetch program stalls until it returns. When it returns, the prefetch controller pushes the object (a fixed number of cache-lines) onto the FIFO between L2 and DL1 and then repeats the process for the next instruction in the prefetch program. Objects pushed to the FIFO wait to reach the head of the queue. Pull and load instructions in the main program then respectively transfer the object to the DL1 and start using it. Data in the FIFO is virtually tagged, and the prefetch controller has access to a private TLB. TLB misses cause the prefetch program to stall just like any other exceptional condition.

Rationale: This design provides a better solution to several issues that are challenging to previous studies. The prefetch program allows the compiler to efficiently encode what must be prefetched, and to handle arbitrarily complex combinations of interleaved spatial prefetching and pointer-chasing. The compiler encodes this information without constraining hardware on when to initiate prefetches, allowing hardware to manage resources better and issue prefetches in a timely manner when resources are free. In practice, the prefetch program is able to run far ahead of the main program. Running ahead is feasible because there is no possibility of cache pollution, and the prefetch program is throttled on a simple condition - when the FIFO fills up. The final transfer between FIFO and DL1 is initiated by pull instructions at the start of loop iterations that specify only how many cache-lines to transfer, not what it must contain. In the common case, this allows the footprint of each iteration to be brought into the DL1 ahead of its use. In the worst case, pull instructions avoid deadlock when the FIFO contains useless data, while limiting the pollution in the cache to a strict upper bound. Prioritization is no longer an issue since the compiler explicitly sequences prefetches.

5.3 The prefetch controller

We now provide a detailed description of the TwoStep microarchitecture in this and the next section, enumerating alternatives and design decisions at important points. We designed TwoStep to be simple, with an orthogonal and parsimonious ISA, while making the compiler's code generation task easier and matching the ISA to common patterns seen in our characterization using DTrack.

We begin with the L2 prefetch controller, the point of origin of each prefetch in TwoStep. The prefetch controller receives two sets of inputs: a prefetch program divided into *kernels*, and initial register values before running a specified kernel. The prefetch program is loaded into the instruction store on application startup, while register initialization is performed under processor control at the start of different program phases. In the rest of this section we assume both program and register values have been initialized, and describe the workflow for a single instruction in the prefetch program. Initialization conditions are specified in the next section.

Table 5.1 describes the ISA of the TwoStep prefetch controller. The instructions in TwoStep's ISA operate on 32 word-length integer registers, one PC register and immediate operands. TwoStep's workhorse instructions are

Fmt	Instruction	Semantics
	arith $\in$ $\{$ add, mul, mod, and, or$\}$	$\circ \in \{+, *, \%, \&, \|\}$
I	arith $R_d, R_s, offset, size$	$R_d \leftarrow R_s \circ offset * 2^{size}$
I	arithp $R_d, R_s, offset, size$	$R_d \leftarrow R_s \circ offset * 2^{size}$; $prefetch R_d$
II	arith2 $R_d, R_s, R_t, size$	$R_d \leftarrow R_s \circ R_t * 2^{size}$
II	arith2p $R_d, R_s, R_t, size$	$R_d \leftarrow R_s \circ R_t * 2^{size}$; $prefetch R_d$
I	load $R_d, R_s, offset, size$	$R_d \leftarrow R_s + offset * 2^{size}$; $prefetch R_d$; $R_d \leftarrow [R_d]$
I	loadp $R_d, R_s, offset, size$	$R_d \leftarrow R_s + offset * 2^{size}$; $prefetch R_d$; $R_d \leftarrow [R_d]$; $prefetch R_d$
II	load2 $R_d, R_s, R_t, size$	$R_d \leftarrow R_s + R_t * 2^{size}$; $prefetch R_d$; $R_d \leftarrow [R_d]$
II	load2p $R_d, R_s, R_t, size$	$R_d \leftarrow R_s + R_t * 2^{size}$; $prefetch R_d$; $R_d \leftarrow [R_d]$; $prefetch R_d$
III	jeq $target, R_s, offset$	if $R_s == offset$: $R_{PC} = target$
IV	jeq2 $target, R_s, R_t$	if $R_s == R_t$: $R_{PC} = target$
III	jlt $target, R_s, offset$	if $R_s < offset$: $R_{PC} = target$
IV	jlt2 $target, R_s, R_t$	if $R_s < R_t$: $R_{PC} = target$
III	jle $target, R_s, offset$	if $R_s <= offset$: $R_{PC} = target$
IV	jle2 $target, R_s, R_t$	if $R_s <= R_t$: $R_{PC} = target$
	next	$++FIFO.tail$

Instruction formats (24-bit instructions):

I	Opcode (5)	R_d (5)	R_s (5)	$size$ (3)	$offset$ (6)
II	Opcode (5)	R_d (5)	R_s (5)	$size$ (3)	R_t (5)
III	Opcode (5)	R_s (5)	$target$ (8)		$offset$ (6)
IV	Opcode (5)	R_s (5)	$target$ (8)		R_t (5)

Field details:

Field	Width (bits)	Encoding	Addressing mode
R_s, R_t, R_d	5	Unsigned	Register
$offset$	6	2's complement	Immediate
$size$	3	2's complement	Immediate
$[x]$	-	Unsigned	Indirect

Table 5.1: The ISA for TwoStep's prefetch controller.

C statement	TwoStep equivalent
`++i;`	add $R_i, R_i, 1, 0$
`c = a + b;`	add2 $R_c, R_a, R_b, 0$
`c = &Arr[a];`	add2 $R_c, R_{Arr}, R_a, 0$
`int Arr[]; c = Arr[a];`	load2 $R_c, R_{Arr}, R_a, 2$ $//2^2 ==$ `sizeof(int)`
`int* Arr[]; c = Arr[a];`	load2p $R_c, R_{Arr}, R_a, 2$
`c = &a` $\rightarrow$ `fld;`	addp $R_c, R_a, offset(fld), 0$
`c = a` $\rightarrow$ `fld;`	load $R_c, R_a, offset(fld), 0$
`Obj* c; c = a` $\rightarrow$ `fld;`	loadp $R_c, R_a, offset(fld), 0$

Table 5.2: Some common access patterns translated into the TwoStep ISA.

of two major varieties: arithmetic and load instructions. Both have a uniform format:

$$Op \;\; R_d, R_s, f, size \tag{5.1}$$

Each instruction scales f by an object-size factor 2^{size}, combines the result with register R_s, and stores the result in register R_d. f may be either a second register (R_t) or a signed immediate operand (*offset*). There are five varieties of arithmetic operations: arithmetic addition, multiplication, and remainder; and logical conjunction and disjunction. Subtraction is provided using negative offsets, while logical left- and right-shifts are provided using positive and negative $size$ exponents, respectively.

Accessing main memory is the fundamental goal of TwoStep, and the ISA provides two major ways to prefetch data into the L2. The first is the load instructions, which act like the corresponding add instruction using indirect addressing. Indirect addressing is implemented by issuing an L2 cache-line-

```
    // R1: root
    // R2: value being searched.
loop:
    jeq continue, R1, 0
        load R3, R1, node_value, 0      // R3 = R1->value;
        jeq continue, R3, R2
        jlt else, R3, R2
then:
            load R1, R1, node_left, 0   // R1 = R1->left;
            jeq loop, R1, R1 // unconditional
else:
            load R1, R1, node_right, 0  // R1 = R1->right;
            jeq loop, R1, R1
continue:
```

Figure 5.2: A simple TwoStep kernel to perform binary search.

aligned prefetch to main memory if necessary, waiting for the prefetch to return, and then performing a simple copy from L2 into R_d. Second, arithmetic and load instructions both have variants — denoted by the p suffix — that prefetch the contents of R_d from main memory after computing R_d. These two techniques are combined in the loadp instruction, which performs a simple add, prefetches R_d, performs the recursive indirect access $R_d = [R_d]$, and prefetches R_d again. These steps are performed serially, and each step waits for prefetches to finish executing before proceeding to the next step. All prefetches are performed on virtual addresses; in our experiments, we use a physically indexed physically tagged (PIPT) L2 cache, and we therefore provide the prefetch controller with a TLB for translation. TwoStep prefetches are treated just like demand fetches because of their near-perfect accuracy — they are not prioritized differently, and they are fetched into the most recently used (MRU) way of the L2. Table 5.2 summarizes the different varieties of prefetches possible in the TwoStep ISA by mapping them to high-level C access patterns. For example, addp corresponds to strided prefetch, while loadp corresponds to pointer prefetch. The difference between add/load and addp/loadp is primarily whether the destination opcode is a pointer that is dereferenced in the current kernel.

In addition to arithmetic and load instructions, the TwoStep ISA contains two additional instructions: control instructions and the novel next instruction. The control instructions are straightforward, consisting of two varieties of conditional branch to *target* depending on comparison between the two operands. The next instruction is used for flow control and explained in the next section. Figure 5.2 shows a simple prefetch program with a single kernel — to perform binary search.

5.4 Flow control: pull and next

The prefetch controller in the previous section prefetches only to L2 and can run arbitrarily far ahead of the main program on the processor, increasing the risk of cache pollution. In order to address both drawbacks, we add a FIFO structure between DL1 and L2, with a width of one DL1 cache-line. Every instruction in the TwoStep ISA knows how many cache-lines it will prefetch and only begins execution if there is room for an equivalent number of DL1 cache-lines in the tail of the FIFO. Cache lines in the head of the FIFO are consumed by pull instructions in the main program, which consume cache-lines from the head of the FIFO and transfer them into the MRU ways of the DL1, causing evictions as necessary. The first cache-line returned by a Pull instruction takes 4 cycles, and every subsequent cache-line takes 1 cycle.

The effect of the pull instruction on flow control is non-trivial. The obvious option is to give a pull instruction the format pull x, where x is an immediate operand. However, such an approach implies that the number of cache-lines associated with a loop iteration must be a static constant. Every prefetched loop must have the same cache footprint along all paths. There are two ways to maintain this invariant:

1. Insert extra pulls at each branch of conditionals with unbalanced footprints. This approach introduces significant overhead in the instrumented application since nested conditionals are extremely common. We quickly discarded this option.

2. Rely on the compiler to count footprints along different paths, to insert the largest possible footprint for a loop, and to insert padding push instructions (addp $< recentregister >, 0$) into some paths of the prefetch program. This approach causes extra overhead in the prefetch program; as we show later, this overhead is not significant. However, it also causes unnecessary pulls throughout an application, and that significantly impacts the latency of pulls into DL1. Another major drawback is the

increase in compiler complexity necessary to track footprints for each path in a loop iteration.

Since neither option is effective, we convert the pull instruction to take no opcodes but instead maintain the count of cache-lines to pull in hardware. Our hardware for maintaining pull counts consists of two pieces: a second, *count FIFO* to maintain count information, and the next instruction in the TwoStep controller ISA. Every push to the main FIFO from the prefetch program increments the counter at the head of the count FIFO, while next instructions at the start of every loop iteration in the TwoStep prefetch kernel bump up the pointer to the tail of the FIFO, creating and initializing a new count. Pull instructions now read the head of the count FIFO to determine the number of cache-lines to transfer. The space overhead for this enhancement is minor, a few bits for every cache-line of FIFO capacity ($<$ 32 bytes in the baseline case). There is no time overhead since the compiler guarantees the count to be at least 1, and reading the count FIFO can be overlapped with the transfer of the first cache-line.

Abnormal situations: So far we have addressed the common case in the execution of a prefetch kernel: the prefetch kernel spends less time per iteration than the main program and thus keeps the FIFO occupied. Periodically the FIFO fills up and causes the prefetch program to stall until there is room. There are two abnormal exceptions to consider: when the prefetch thread generates invalid prefetches, and when it falls behind the main program. The challenge in each case is first to maintain synchronization between main and prefetch programs, and second to avoid polluting the cache. Prefetches to invalid addresses do not stall the prefetch thread; instead the prefetch thread inserts invalid cache-lines into the FIFO in order to maintain synchronization. When the prefetch thread falls behind the main program the FIFO empties out. Subsequent pulls increment a counter when they are unable to pop items off the FIFO. The counter provides the prefetch program with some slack to catch up with the main program, as future calls to next prefetches decrement the counter rather than push items on the pull-count FIFO. If the counter drops back to zero the prefetch thread can start pushing items onto the FIFO again. If the counter instead saturates to some maximum level, usually FIFO capacity, the prefetch thread is aborted.

5.5 Maintaining coherence

TwoStep maintains a copy of a program's data in the FIFO; it is possible for this data to become stale in some situations. For example, consider a scenario where the main program fetches, writes to and and evicts a cache-line from the DL1 between the time that cache-line is pushed into the FIFO by the prefetch controller and the time it arrives at the head of the FIFO and is transferred to the DL1. The main program could now end up reading stale data.

Handling coherence requires mechanisms and policies for detection and recovery. There are two broad techniques to detect a coherence conflict be-

Instruction	Semantics
pull	Transfer cache-lines from FIFO to DL1 as described in Sections 5.4 and 5.5.
rcopy $R_d \leftarrow R_p$	Copy the contents of processor register R_p to TwoStep register R_d
start pc	Copy immediate field pc into TwoStep PC register.

Table 5.3: ISA extensions for the main general-purpose processor.

tween cache and FIFO: first, scan the FIFO for duplicates when pushing, and second, to scan the FIFO for duplicates when pulling. Similarly, recovering from a conflict presents two options: either flush the FIFO, invalidating all its contents without changing FIFO size in order to preserve synchronization, or invalidate conflicting cachelines. Both detection and recovery can be speeded up by using a hardware hash-table for filtering checks. Using such a hash-table implementation implies that search is fast, and therefore invalidating just conflicting cachelines is uniformly preferable to invalidating the entire contents of the FIFO. Later in this chapter we examine the effects of coherence conflicts on the benefits of TwoStep in an idealized manner, without commenting further on the low-level mechanisms for coherence detection and recovery.

5.6 Initializing registers before kernel execution

We conclude our description of TwoStep with a description of the procedure for initializing a prefetch thread and activating it. Table 5.3 summarizes the extensions to a general-purpose processor ISA required by TwoStep. Design decisions behind the pull instruction has already been covered in detail.

In addition, the processor requires two types of instructions to setup and kick off prefetch programs for different program phases. The first is rcopy to copy processor registers into their counterparts in the L2 controller, supplying the prefetch kernel with all necessary inputs. After some number of rcopy instructions, the main program then executes a start instruction to set the PC register of the L2 controller and commence prefetch kernel execution. Overheads in these latter two instructions are easily tolerated; in our implementation, each rcopy and start instruction takes up 10 instruction slots in the main processor pipeline without impacting prefetch thread performance. This overhead should be a conservative estimate of the most likely implementation for these instructions in a production setting — using memory-mapped I/O.

5.7 Interactions between pulls and stock compilers

One issue arose in our implementation because we choose to instrument the main program at the level of the source code just like with DTrack, rather than in the binary. As a result, pull instructions within loop nests can perturb the code a conventional compiler generates. Since pull instructions occur in the inner loops of the application, any such perturbance is likely to cause significant degradation in performance. Since the Alpha compiler we use is not aware of their semantics, this encoding has changed several times to work around idiosyncracies in optimization policies. Prior versions of the pull instruction caused the compiler to suppress loop unrolling and software pipelining for tight loops containing pull instructions. Our current version maintains pointers to

Feature	Size/Value
#Registers	32
Instruction store	2KB
FIFO capacity	2KB
Pull latency	4 for first cache-line
	1 cycle for subsequent cache-lines
Prefetch controller TLB capacity	Infinite

Table 5.4: Baseline TwoStep configuration. Processor configuration in Table 3.1.

each of the memory-mapped addresses used for instrumentation, in order to keep the compiler from hoisting these loop-invariant stores out of the loop they are intended for. In a production setting the compiler's policies will have to be modified to ignore pull instructions.

5.8 Experimental Methodology

In order to assess the feasibility of TwoStep, we evaluate it over 8 of our applications in the rest of this disseration. Benchmark choice was largely driven by the characterization detailed in Chapter 3: 300.twolf, sphinx, and 181.mcf are irregular applications with the most intensive traffic to memory; 183.equake is a regular memory-intensive application; 179.vpr and 188.ammp are irregular applications with moderate memory traffic; finally, 164.gzip and 179.art are regular applications with low memory traffic. This chapter's initial exploration using hand-crafted prefetch kernels further focusses on just 4 of these applications: 179.art, 181.mcf, 300.twolf, and sphinx. We run these applications on a version of sim-alpha [25] enhanced with an implementation of

TwoStep prefetching. Hints are used to implement pulls as well as demarcate the endpoints of each simulation interval in terms of high-level loop iterations. We specify high-level simulation start- and end-points for each application in order to make consistent measurements across different binaries with and without pull instructions. Both baseline and transformed codebases are compiled with the aggressive Alpha GEM cc compiler [75]. Table 3.1 earlier summarized the baseline demand-fetched machine configuration; Table 5.4 now enhances this configuration with a baseline TwoStep configuration, specifying the size of the instruction store, the default FIFO capacity, Pull latency, and TLB capacity. Sensitivity results at various points in the next 3 chapters will motivate these design choices.

Selecting a baseline machine configuration: Our baseline includes no prefetching in the data caches. This decision was made for two reasons:

1. Neither the Alpha 21264 nor most past literature on prefetching included hardware prefetching in the baseline. By following precedent, we allow convenient comparison with prior work.

2. Not all prefetch schemes can be favorably combined with each other. Subtleties in the design of different prefetch schemes affect interactions between them. By using a purely demand-fetched baseline, we avoid favorable or unfavorable perturbations to our results. This approach allows us to safely explore interactions with other prefetch schemes in Chapter 7.

Comparing TwoStep with other prefetch techniques: We now briefly outline our methodology for comparisons with other prefetch techniques, both using hand kernels in the rest of this section, and using the TwoStep compiler in Chapter 7. The TwoStep compiler is based on C-to-C translation using the C-Breeze compiler toolkit [30], coupled with the same optimizing Alpha GEM cc compiler in the backend. Our major comparisons are with Tagged prefetch [87] and a family of region prefetching techniques: Scheduled Region Prefetching (SRP) [55] and Guided Region Prefetching (GRP) [99].

Tagged prefetch prefetches the next cache-line on an L2 cache miss, and it marks cache-lines as prefetches using an extra *tag* bit to mark non-speculative data. This bit is set for demand fetches on initial fetch, and for prefetches on their first non-speculative use. This approach allows limited lookahead and concomitant improvement for simple spatial patterns, but fails to improve more irregular applications.

SRP consists of a scheduler at the L2 that prefetches data from memory in 4KB-aligned *regions* around addresses causing cache misses. The flow of prefetches is tuned to not slow down the processing of demand fetches; demand fetches are prioritized over prefetches in the cache hierarchy (old and unprocessed prefetches are silently dropped), and prefetches are placed in the LRU way of the L2 to reduce cache pollution for applications with irregular access patterns. GRP augments these region prefetch mechanisms with compiler-generated hints for pointer as well as region prefetching that serve to improve accuracy and eliminate region prefetching in irregular applications.

The techniques we compare TwoStep with span the spectrum from the state of the art in production hardware to the state of the art in research prototypes. Tagged prefetch is a simple hardware mechanism that exemplifies mechanisms included in many production processors. As such, it provides a common baseline of production machines to compare against. We selected GRP and SRP as our examples of more recent research for three reasons. First, we wanted the techniques we compare with to be relatively recent, and reasonable exemplars of the state of the art, showing sophisticated decisions for prefetch selection, timing and pollution-avoidance. Second, we wanted a broad coverage of both hardware and software techniques, and of techniques addressing both spatial and pointer prefetch. Third, we were constrained by methodological constraints of easily-accessible infrastructure. Choosing a family of techniques allows us to perform comparisons across just two parallel compiler-simulator toolchains — *C-Breeze+TwoStep+sim-alpha* and *Scale+Region prefetch+sim-outorder* [99] — thereby cutting down on our infrastructure-management overhead and also on the baselines we need to track. While the machine configurations are largely the same, GRP and SRP use the `sim-outorder` microarchitecture to run Alpha ISA binaries [9] rather than the detailed model of the Alpha 21264 that we use [25]. In addition, GRP is compiled for the Alpha ISA using the Scale research compiler [62].

Feature	181.mcf	300.twolf	179.art	sphinx
Prefetch program size (1-byte instructions)	52	37	29	100
Cache-lines pulled per inner loop iteration	3-12	2-11	1	1-7
# Phases per topmost iteration	3	1	1	5
# Distinct loop nests	8	1	7	1
Max nesting depth	2	3	2	3

Table 5.5: Vital statistics of our hand-crafted prefetch programs

5.9 Preliminary evaluation with hand-crafted prefetch kernels

This section summarizes some initial findings of our study, using hand-crafted prefetch kernels to evaluate TwoStep. We begin with hand-crafted kernels for two reasons. First, they allow us to explore the potential of our approach independent of compiler implementation. These results were generated before the completion of the compiler implementation as a feasibility study. Second, our hand-crafted kernels act as benchmarks for the later compiler implementation, and subsequent chapters will show that we do well at fulfilling the potential of TwoStep even though the compiler-generated kernels are very different.

Our findings are in two categories. First, we evaluate TwoStep and show significant speedups for the irregular applications we selected. Second, we perform various sensitivity analyses in the design space, compare TwoStep with some prior prefetching studies, and analyze our improvements by data structure to confirm our intuitions. Table 5.5 highlights the small size of our

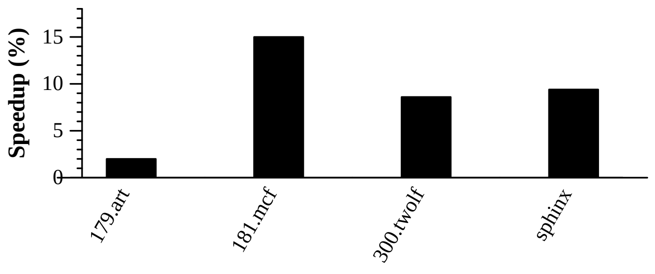


Figure 5.3: Improvements with an infinite FIFO

hand-crafted prefetch programs and the small footprint of loop iterations, as measured by the number of cache-lines pulled in each. Our detailed characterization of the previous chapters now yields a small number of distilled prefetch kernels that provide substantial prefetch coverage in just 1-8 loop nests with less than 100 instructions in the TwoStep ISA, each nest at most 3 loops deep.

Measuring limit performance: We begin by measuring the performance of TwoStep relative to the baseline. For this experiment, we configure TwoStep with an infinitely long FIFO so that the prefetch engine never has to stall to wait for the main program to catch up. Pulls have a latency of 4 cycles between request from FIFO and transfer to DL1. Figure 5.3 summarizes the reduction in total cycle time after simulating well-defined intervals of our application with TwoStep enabled. TwoStep shows speedups of between 10 and 15% for our 3 irregular applications. The regular application 179.art has more minor speedups, hinting at TwoStep's limitations. We examine more applications in Chapter 7 to determine the extent of this issue, and to investigate its causes.

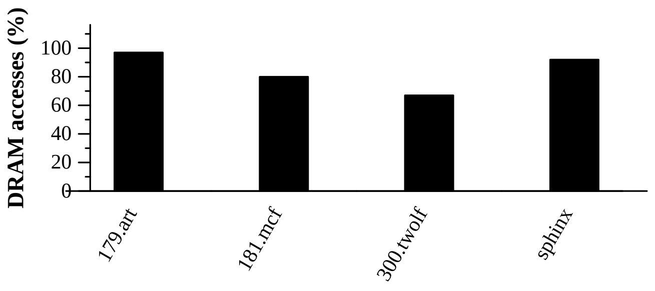


Figure 5.4: Fraction of main memory accesses remaining

Figure 5.4 demonstrates a second strength of TwoStep: we show that successful prefetching may be accompanied by *reductions* in the number of accesses to main memory. While most prefetching studies at best avoid increasing aggregate bandwidth requirements to main memory, the high accuracy of TwoStep prefetches allows cache-lines to turn dead after their last prefetch in an interval. This compression of live times increases temporal locality, resulting in reductions in DRAM access counts. These initial results establish the promise of TwoStep: accurate and well-timed prefetching into the cache hierarchy for arbitrarily irregular access patterns.

Prefetching effectiveness: We now analyze the results of Figure 5.3 more closely in order to understand the source of our speedups. In spite of the reductions in cycle count, the number of DL1 misses is relatively unaffected by TwoStep. To gain a deeper understanding of the critical path, we track cycles that the pipeline commits no instructions, assigning blame to the data structure of the load at the head of the reorder buffer. Figure 5.5 summarizes

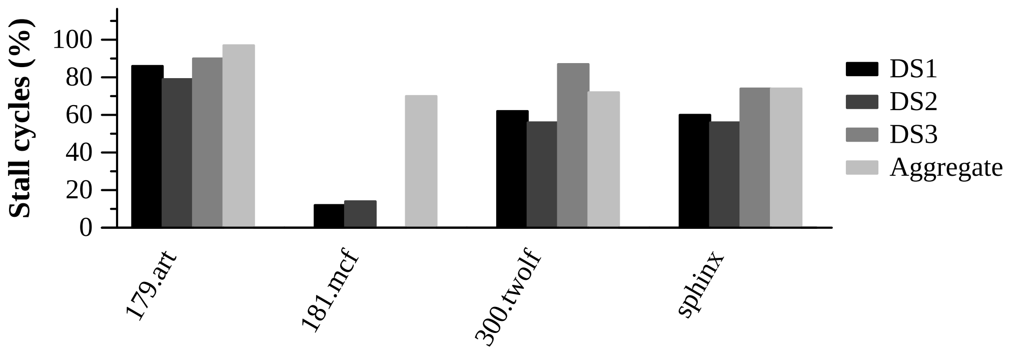


Figure 5.5: Stall cycles remaining after TwoStep prefetching for the most frequently missing data structures (DS1 and DS2), compared to reduction in aggregate stall cycles due to memory

the number of stall cycles reduced for each application on a data structure basis. We show 4 bars for each application in this figure, for the top 3 data structures by miss-count (the same data structures as in Tables 3.3–3.5), and for the application in aggregate. Each bar shows the percentage of stall cycles remaining after TwoStep prefetching is applied to the baseline machine configuration. Figure 5.5 shows that pipeline stalls due to major data structures DS1 and DS2 are reduced. These are the data structures targetted by our prefetch programs. The impact of these reductions on aggregate pipeline stalls due to memory is, however, markedly lower. We believe that understanding the precise reasons for this difference — the other data structures that are now critical — will be a fruitful avenue for future research.

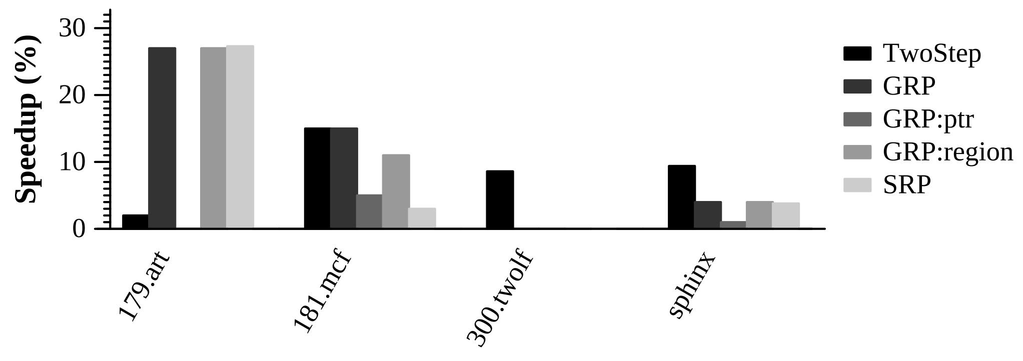


Figure 5.6: Comparison of TwoStep with some prior prefetching studies.

Comparison with prior studies: Having performed a detailed comparison of TwoStep with a no-prefetch baseline, we now compare TwoStep with a family of region prefetching techniques from prior work. As detailed in the previous section, our results for region prefetching were obtained on a parallel toolchain to ours; we therefore compare their speedups relative to independent baselines.

Figure 5.6 summarizes the results of our initial comparison. For each of our initial applications, we show the reduction in cycle counts resulting from TwoStep and 4 region prefetch setups: GRP, GRP with only pointer-prefetch hints enabled, GRP with only region prefetching hints enabled, and SRP which provides no hints. TwoStep does substantially better than all these approaches for 300.twolf and sphinx, and as well as the best of them for 181.mcf, but substantially worse for 179.art. Thus, both GRP and SRP have poorer coverage than TwoStep among irregular applications, but provide substantially better performance for regular applications. We return to these

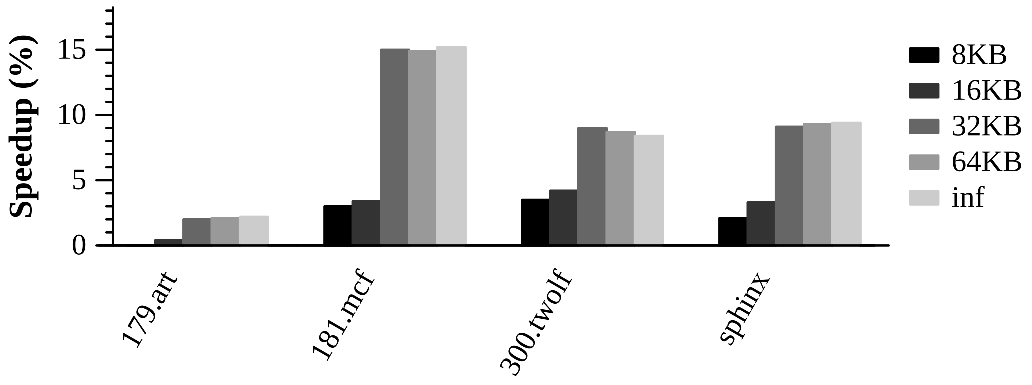


Figure 5.7: Sensitivity of speedups to FIFO capacity

bipolar results for a more detailed study in Chapter 7.

SRP's performance largely matches that of GRP, but with lower prefetch accuracy and more profligate use of main memory bandwidth. However, the benefits of spatial and pointer prefetching do not follow superficial trends. 181.mcf and sphinx are almost purely pointer-based techniques, but are improved more by the region prefetching in GRP than the pointer prefetching. These phenomena arise from accidental interactions with the memory allocator. In Chapter 7 we return to them and argue that such accidental interactions are easily lost due to experimental changes such as a larger input set.

Sensitivity analysis: The TwoStep design has two major parameters - FIFO capacity and pull latency - that must be realistic in order for it to be feasible. We now evaluate its sensitivity to these parameters. Figure 5.7 summarizes the speedups obtained by TwoStep for our applications and the

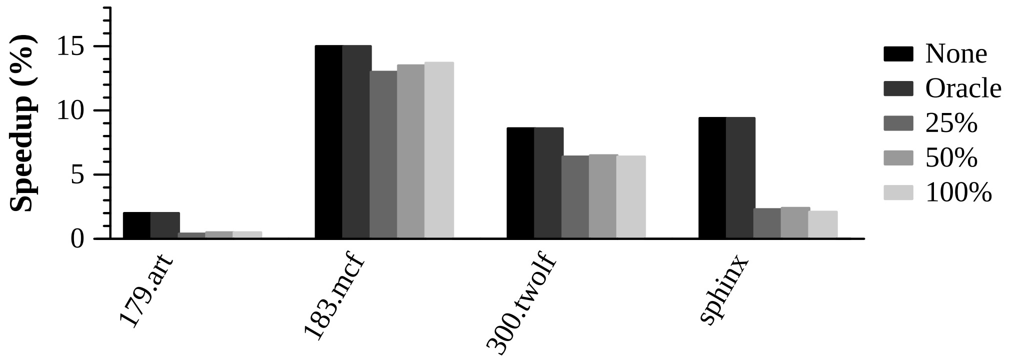


Figure 5.8: The effect of coherence conflicts on performance. Percentages indicate fraction of false positives

sensitivity of these improvements to FIFO capacity. A 2KB (32-entry) FIFO suffices to provide most of the benefit of an infinite-capacity FIFO, indicating the effectiveness of the FIFO at realistic capacities for current technologies [2].

Coherence: We now evaluate the effect of handling coherence issues between the caches in FIFO in TwoStep. Correctness is not affected as our timing-based simulation model is independent of the model of functional computation. We consider an oracle implementation that decides whether to pull or discard each cache-line in the FIFO based on prior stores to that address. We then randomly insert false positives in the oracle's decisions in order to gauge the sensitivity of our speedups to conflicts in the FIFO due to coherence. Figure 5.8 shows that coherence with an oracle degrades our speedups by less than 1%. Performance degradation is negligible upto 50% false positives (i.e. half the FIFO entries are invalidated on each store). These results show our scheme to be robust

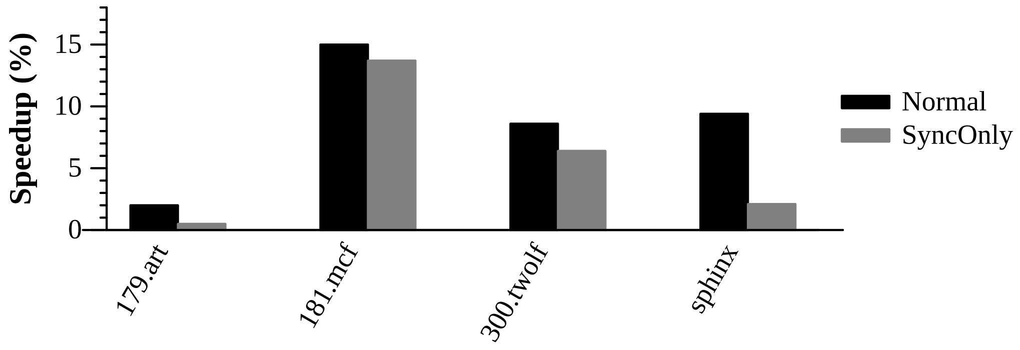


Figure 5.9: The importance of pushing to DL1

to coherence conflicts, primarily due to the relative infrequency of stores in our applications. The case of 100% false positives we consider in more detail below.

Prefetching to L2 vs DL1: Figure 5.9 attempts to tease apart the twin benefits of TwoStep - prefetching data to the L2 and making it available at the DL1. It compares two configurations:

1. *Normal*: A conventional TwoStep microarchitecture.
2. *SyncOnly*: A modification to TwoStep where pulls remove cache-lines from the FIFO but do not transfer them to DL1.

In the latter case, the FIFO acts purely as a synchronization mechanism, causing the prefetch controller to stall when it runs too far ahead of the main program. It is behaviorally very similar to the case of coherence with

100% false positives (always flush FIFO on store), and our results for these two configurations are identical. Figure 5.9 shows that the importance of pushing data to the DL1 varies by application; 181.mcf and 300.twolf derive more than 75% of their speedups from L2 prefetch, while in sphinx and 179.art more than 75% of the speedups is derived from prefetching to the Dl1. We explain these results in more detail in Chapter 7.

5.10 Summary

Irregular applications contain sophisticated access patterns. TwoStep prefetches for such applications by providing simple hardware mechanisms - a prefetch engine and a FIFO - that can be controlled by software. The hardware mechanisms have useful properties: fewer constraints on prefetch scheduling, resistance to DL1 pollution, and easy throttling. These improvements are achieved at the cost of some burden to software: the compiler must statically map prefetches in the prefetch program to pulls in the main program, and ensure that the two stay synchronized. Initial experiments with hand-crafted kernels show that it performs as expected for irregular applications, but not as well for relatively regular applications. We now describe the compiler-side component of this thesis before generating results for more applications and identifying more rigorously the high-level characteristics that influence application synergy with TwoStep.

Chapter 6

Compiler support for TwoStep

This chapter describes and evaluates compiler algorithms to generate useful precomputation kernels for TwoStep. Our compiler is structured to convert from C to C, outsourcing back-end optimizations to an off-the-shelf C compiler. It uses information from an interprocedural pointer analysis, and performs several context-sensitive traversals of the whole program, starting at the beginning of `main()` and processing function bodies everytime a call to them is encountered.

We begin by enumerating the requirements for such a compiler, then use these requirements to drive a staged tour of the compiler as a series of refinements from the top down (Figure 6.1). The major challenge in designing the compiler is to manage overheads due to pull instructions in our major loops. A purely brute-force approach that tries all possible combinations of the major loops is infeasible; instead we stage information from different sources — loop profiles and slice densities — to perform feedback-based backtracking in the search space of loop nest combinations. Figure 6.1 reflects this back-tracking oriented architecture, described it in detail in Sections 6.1–6.4.

After the description, we contrast our compiler to the major prior work

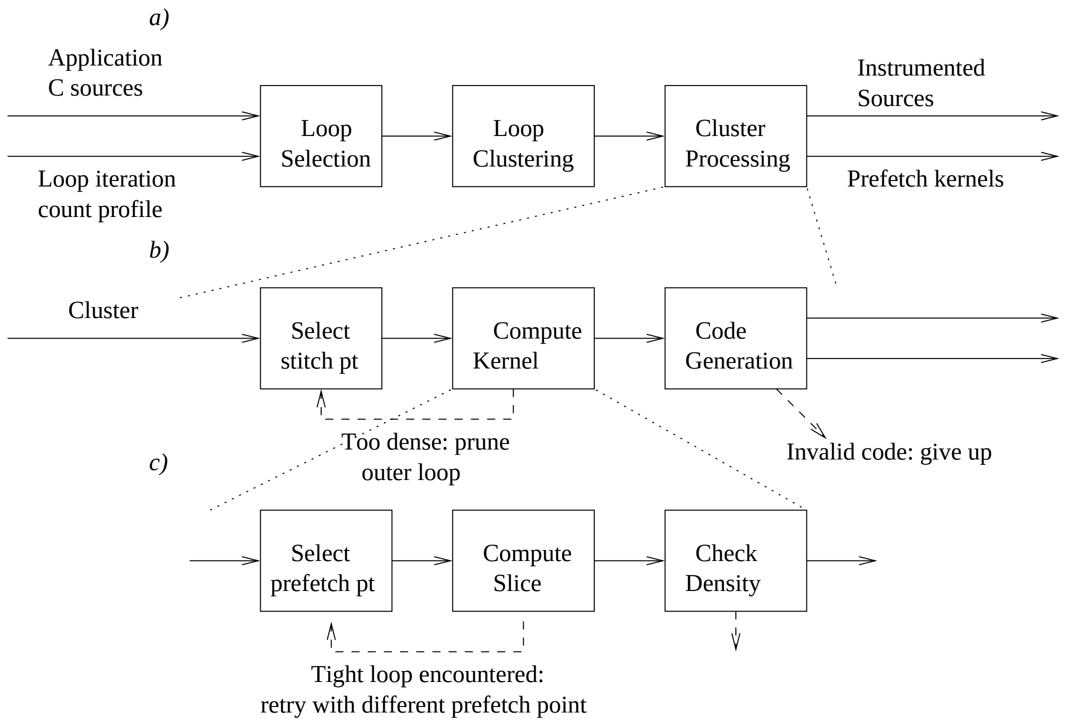
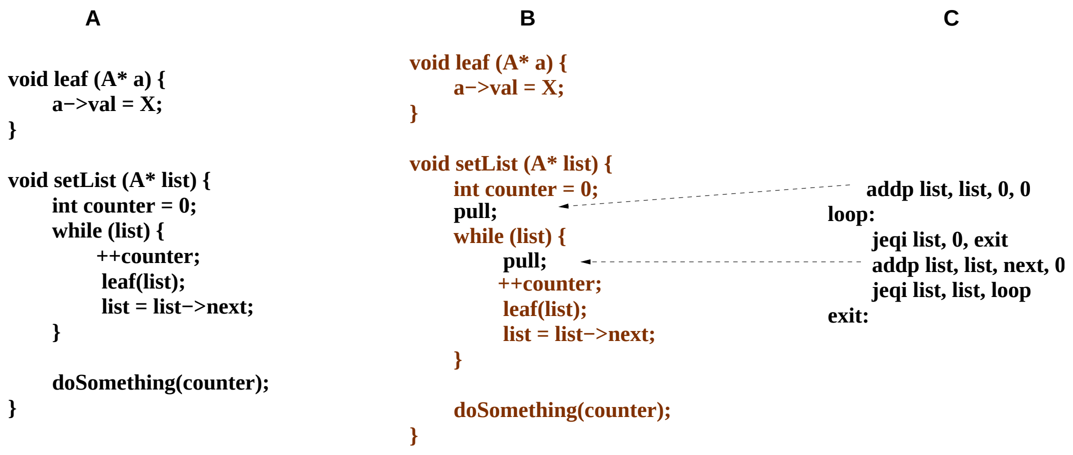


Figure 6.1: Overview of the TwoStep compiler as a series of refinements.



```
void leaf (A* a) {
      a->val = X;
}

void setList (A* list) {
      int counter = 0;
      while (list) {
            ++counter;
            leaf(list);
            list = list->next;
      }

      doSomething(counter);
}
```

```
void leaf (A* a) {
      a->val = X;
}

void setList (A* list) {
      int counter = 0;
      pull;
      while (list) {
            pull;
            ++counter;
            leaf(list);
            list = list->next;
      }

      doSomething(counter);
}
```

```
      addp list, list, 0, 0
loop:
      jeqi list, 0, exit
      addp list, list, next, 0
      jeqi list, list, loop
exit:
```

Figure 6.2: A simple C program (A), pull instructions added to it (B), and the corresponding prefetch program (C). Arrows connect prefetches in the prefetch program with corresponding pull instructions in the main program.

in compiling for precomputation and enumerate the major areas where precomputing for TwoStep presents a different set of contraints than compilers have faced in the past. Finally, we perform a comprehensive offline validation of our compiler's policies, exploring the entire state space for our applications in search of good slices that may have been missed. This analysis provides insight into one limitation of precomputation-based prefetching: when prefetch bandwidth utilization is critical in tight loops, it is necessary to trade off prefetch coverage for slice density. Slices that are too dense result in prefetch kernels that do much of the same work as the main processor, reducing the prefetch thread's ability to run ahead of the main program and therefore its effectiveness.

6.1 Goals and requirements

Figure 6.1 illustrates the transformations TwoStep requires. Given application C sources it must emit useful prefetch kernels in the TwoStep ISA at the L2 controller, and appropriately instrument the *main program* binary running at the processor. These twin modifications require mechanisms and policies for the following:

1. Selecting loads most likely to cause pipeline stalls. We call these static program locations *prefetch points.*
2. Selecting for each prefetch point a *stitch point* — a location where precomputation may profitably be started, early enough to give TwoStep the slack necessary to run ahead, but not so early as to cause the prefetch program to grow too bloated, or to be often led astray before the prefetch point is reached.
3. Generating the prefetch program corresponding to all the computation necessary to compute the prefetch point from the stitch point.
4. Inserting pulls at the start of each loop involved.

For an illustration of these transformations, see Figure 6.2. This figure shows a simple program to operate on a linked list, the places where the compiler needs to insert pulls, and the corresponding prefetch program to run on TwoStep. We use this example at various points in the rest of this chapter. The crucial requirements for the compiler are to generate prefetch programs shorter than the corresponding parts of the main program so that it runs ahead, and for the instrumentation in the main program to be lightweight. Also, every pull executed by the main program must do useful work to justify its overhead; the compiler must avoid inserting pulls at locations where the prefetch program is unlikely to have data in the FIFO. In the next three sections, we describe the process by which the TwoStep compiler meets these requirements.

6.2 Analyzing the application by loop cluster

Given the above requirements, the compiler's flow can be decomposed at the highest level into 3 pieces as shown in Figure 6.1 *a*): Loop selection to identify what must be prefetched, loop clustering to maximize slack for the prefetch kernel, and cluster processing to generate at most one prefetch kernel per loop cluster. Our first step, loop selection, uses one piece of easily-obtainable profile information — loop counts. To statically compute the *dereference volume*:

$$DV_{loop} = Iters_{loop} * StaticPtrs_{loop} \tag{6.1}$$

where $Iters$ is the average number of iterations of this loop per loop entry, and $StaticPtrs$ is the path-insensitive count of deref operations in this loop body *excluding* loops nested within it. We then sort the list of loops by DV, shortlist the top loops that add up to 90% of total application DV, and commence the second step — loop clustering.

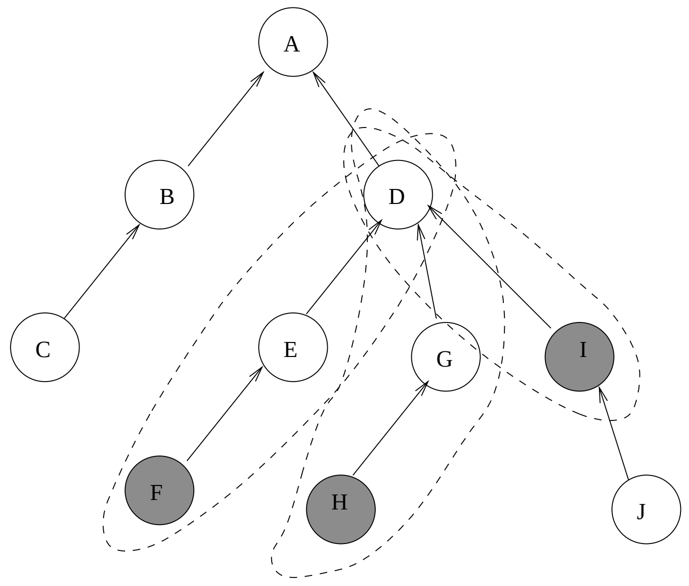


Figure 6.3: Application viewed as a tree of context-sensitive loops. Shaded nodes are shortlisted loops. Three clusters are shown. Leaf C does not belong to a cluster because it has no ancestor in the shortlist.

Loop clustering: Figure 6.3 depicts the compiler's view of an application during loop clustering. The application is a tree of loops. The root node represents the entire application — the body of the `main()` routine, and all other nodes represent context-sensitive loops. The children of each node are loops contained within its body. Shortlisted loops are shaded. Function boundaries have been elided. The compiler builds up loop clusters starting at each leaf loop — a loop with no subordinate loops — by scanning outward adding container loops to the cluster, occasionally creating a boundary and starting a fresh cluster. Our clusters maintain the following invariants:

- Each cluster contains innermost exactly one shortlisted loop. If we encounter a second we start a new cluster. Leaves without an enclosing shortlisted loop at any level are discarded, since they provide insufficient incentive for prefetching.

- Each cluster contains as many loops as possible both above the shortlisted loop. Later phases may subset a cluster; we give them as much to work with as possible.

- We never allow a cluster to grow past a function boundary if it is called in multiple contexts in the loop tree and not all of these contexts lie within a cluster. More precisely, we permit a loop A to be added to the cluster of a subordinate loop B in a different function f only if 99% of iterations of loop B (from the loop profile) have an ancestor in a cluster.

  This condition prevents us from adding the overhead of pulls in contexts where there will not be a prefetch kernel running any significant fraction of the time. Enforcing this condition requires a second pass after clustering to prune bad clusters.

The rest of the compiler processes these clusters in descending order of their DV.

$$DV_{cluster} = \sum DV_{loop} \tag{6.2}$$

This ensures we prioritize our cluster candidates by expected cache miss count. The list of clusters can have overlap and usually does; after a cluster is successfully processed no member or ancestor loop can be processed again. This constraint prunes some clusters and eliminates others entirely from consideration.

6.3 From loop cluster to prefetch kernel

As depicted in Figure 6.1 *b*, processing a cluster consists of accepting a cluster of loops as input and emitting at most one prefetch kernel corresponding to it. It consists of three major phases — stitch point selection, kernel computation, and code generation — and one feedback path to prune successive outer loops from a cluster if the resultant prefetch kernel is found to be too dense relative to the main program. We now focus on the first and third, postponing the description of the kernel computation to the next section.

Stitch-point selection: Given a cluster of loops, the stitch point is the point in the program to insert *stitch code* to trigger the corresponding prefetch kernel. Since stitch code must trigger precisely once for every execution of the loop cluster, a good stitch point has the following properties:

- As a boundary-condition initialization, it occurs outside the loop cluster itself.
- It dominates the cluster; every execution of the cluster should have executed stitch-point code.

- It does not lie outside the loop containing the cluster. Stitch code must execute *every time* the cluster is entered.

- It does not lie before a sibling loop in the loop tree. This prevents too-early initialization as well as destructive overlap between prefetch kernels.

- It does not lie before a sibling function call. Again, this prevents arbitrary gaps between initialization and prefetch use. However, this constraint does not exclude the possibility of the stitch point and cluster being in different functions; the stitch point may lie further up the call stack subject to previous constraints. If we span a function boundary, however, we must compute a good stitch point in every possible context of the function.

- It occurs as far before the cluster as possible subject to the previous constraints.

Figure 6.4 shows our algorithm for selecting good stitch points, taking these constraints into account. The individual conditions have a one-to-one correspondence with the above properties. We add two points to clarify the recursive case when moving the stitch point up the call stack. First, we can clear `answerStack` because we are guaranteed to find at least one more dominating statement where the stitch code may be inserted — right before the last call. Second, the recursive call cannot be passed `cluster` itself; it must instead be

```
// l is the context-sensitive statement list of the input program

selectStitchPoint(stmt, cluster, answerStack):
    traversing s upwards from stmt in l:
        if s dominates cluster: answerStack.push(s)
        if function call is encountered: break
        if loop boundary is encountered: break
        if function header is encountered
                and there is more than one caller:
            clear answerStack
            for every calling context c:
                cluster' = correspondingContext(cluster, c)
                stitch' = selectStitchPoint(c, cluster', [])
                answerStack.push(stitch')
            end
            return answerStack
        end
    end

    return answerStack.top
end

// Usage: selectStitchPoint(cluster, firstStmt(cluster), [])
```

Figure 6.4: Stitch point selection starting at a specific statement. Takes a loop cluster as input and returns a list/stack of context-independent statements after which stitch code should be instrumented.

a context-sensitive statement corresponding to the static cluster but in the same context as the caller `c`.

Code generation: Once a stitch point is selected and its prefetch kernel computed and found to be not too dense, it remains only to emit the prefetch kernel in terms of the TwoStep ISA. A simple one-pass code generator suffices for this purpose, with simple rules for translating each statement type in a lowered C form — containing only ifs and gotos and no more than one binary operation and one assignment per statement as shown in Figure 6.5 — into some sequence of TwoStep instructions. Our prototype compiler performs no register allocation, assuming an infinite pool of registers. It also performs no back-end optimizations. Later in this chapter, we show that these decisions do not impact our evaluation. The only other complication is the book-keeping necessary to skip past empty basic blocks without perturbing the global control structure of the prefetch kernel.

There are a few rare circumstances where the compiler is currently unable to generate code for a prefetch kernel: if the kernel contains a call to a library routine whose body is not available to our whole-program analysis, or if it contains a recursive function call. In these circumstances we currently discard the kernel. Otherwise, we insert the stitch code computed during slicing (described below) and insert pulls at the start of each loop in the cluster.

```
void setList(A * list) {
    int counter, __T0, __T1, __T2;
    {
        counter = 0;
        goto __L0;
    }
    {
__L0:;
        if (list == 0) goto __L1;
        goto __L3;
    }
    {
__L3:;
        __T0 = counter + 1;
        counter = __T0;
        __T1 = leaf(list);
        list = (*list).next;
        goto __L0;
    }
    {
__L1:;
        __T2 = doSomething(counter);
    }
}
```

Figure 6.5: Linked list traversal in lowered C form.

6.4 Selecting a good slice for a fixed cluster

We now turn to Figure 6.1 *c*), the final component of the TwoStep compiler. Once again, we divide up the process of generating kernels from a loop cluster and fixed stitch point into three phases — prefetch point selection, slice construction, and a density check. Slices that are found to be too dense are retried after stripping an outer loop from the cluster as described in the previous section. One final heuristic is to discard slices that contain a loop with a single basic block, because all such *tight loops* serve to do is to allow the main program to catch up with the prefetch program, without actually providing any prefetching benefit. We perform this test after slicing because in practice such really tight loops are often not part of the slice even if they are within the loop cluster of interest. When we encounter them in a slice we backtrack to pick a different prefetch point and recompute the slice.

Prefetch point selection: The prefetch point of a cluster is a pointer dereference to be prefetched within the innermost loop of the cluster. We simply pick the first such statement we find, checking that it cannot be hoisted out of the innermost loop, and avoiding the innermost-loop inductive variable if possible. We rely upon later checks for slice density to backtrack and try a different prefetch point if necessary.

Slice computation: Given a prefetch point and a stitch point we can now compute the backward slice starting at the prefetch point. Figure 6.6 illustrates the necessary inter-procedural transformation. The slicing algorithm consists of starting at the prefetch point and traversing back the interprocedural reaching-definitions as computed by the pointer analyzer. We mark every statement encountered in this tree traversal, cutting traversal short when we attempt to move to a statement before the stitch point in the context-sensitive statement list of the program (statement `1` in Figure 6.4).

Once the set of statements in the slice is computed, we can identify the set of values that need to be transferred to the TwoStep prefetch controller at the stitch point. We perform a backward interprocedural traversal, adding values on the right-hand side of statements in the slice as we encounter them, and removing values on the left-hand side. When a procedure call is encountered, we rename formal parameters with call arguments and proceed. This traversal contains a parsimonious list of the variables that need to be seeded into TwoStep's registers from those of the main processor before starting the prefetch kernel for the current slice.

Density check: Having computed the slice, we must now check that it prunes enough computation to allow the prefetch thread to run ahead of the main program. Our density metric is the fraction of the *statement volume* between prefetch point and stitch point that is part of the slice.

$$SV_{loop} = Iters_{loop} * SlicedStaticStmts_{loop} \tag{6.3}$$

$$SV_{cluster} = \sum SV_{loop} \tag{6.4}$$

```
void leaf (A* a) {
    a->val = X ;
}

void setList (A* list) {
    int counter = 0 ;

    while (list) {
        ++counter ;
        leaf (list) ;
        list = list->next ;
    }

    doSomething (counter) ;
}
```

```
loop:
    if (!list) goto exit ;
    list->val = X ;
    list = list->next ;
    goto loop ;
exit:
```

Figure 6.6: A simple C program and its context-sensitive interprocedural backward slice

$$TV_{loop} = Iters_{loop} * StaticStmts_{loop} \tag{6.5}$$

$$TV_{cluster} = \sum TV_{loop} \tag{6.6}$$

$$Density_{cluster} = SV_{cluster}/TV_{cluster} \tag{6.7}$$

In these equations, $SlicedStaticStmts_{loop}$ is the number of simple statements in 3-address form in one iteration of the loop that belong to the slice, and $StaticStmts_{loop}$ is the total number of such statements in this iteration. Slices with densities under a fixed threshold of 60% are retried with other prefetch or stitch points as outlined above. Our empirical reasons for selecting this threshold are described in Section 6.5.

Summary: We have described the implementation of the TwoStep compiler in detail. TwoStep transforms an application augmented with loop iteration count profiles into prefetch kernels in the TwoStep ISA for the important loop clusters. Parts of this workflow are common with other slicing and precomputation studies, while parts are necessitated by the novel TwoStep microarchitecture. In the rest of this chapter, we describe a preliminary evaluation of our compiler comparing slices generated automatically with those generated by hand. We then discuss in greater depth the effect of a pull-based prefetching microarchitecture on the compiler and how it differs from prior algorithms for automatic precomputation.

6.5 Evaluating the slices generated by the compiler

This section evaluates each of the major policies in our compiler, and we demonstrate that these policies adequately cover the state space for our applications. We cover in order: loop clustering, densities for different cluster configurations (the backtracking loop in Figure 6.1 *b*), and finally the effect of prefetch point selection on density (loop of Figure 6.1 *c*). We then summarize the vital statistics of the prefetch kernels selected for each of our applications.

Loop clustering: Clustering bounds the state space for searching for useful prefetch kernels in later passes. Table 6.1 summarizes the usual size of this state space, measured as the distribution of loops of different nesting-depths in our applications. These loop nests all contain innermost loops in the top 90%

Application	#Loops of nest depth: 1	2	3	4
175.vpr	8	14	0	0
179.art	0	6	2	1
181.mcf	0	1	1	1
183.equake	0	0	3	0
188.ammp	3	4	5	1
256.bzip2	4	3	2	3
300.twolf	7	8	0	0
sphinx	3	6	0	0

Table 6.1: Size of the clustering state space

of loop volume for the application. Loop nest candidates within an application often have overlapping outer loops; the total loop volume for these nests often exceeds 100%.

Choosing loop clusters: Since a prefetch kernel for one loop eliminates overlapping kernels in any containing loops, the goal is to maximize the loop volume that is covered by kernels without drawing too much computation into the kernel. The density threshold is a crucial parameter in the design of the TwoStep compiler, and affects the ability of the compiler to handle deeply-nested loops. In picking a good density threshold, we are guided by the densities of the most deeply nested loops in our applications, some of which are shown in Table 6.2. In this figure, we assume prefetch point selection as described in Section 6.4 and study the effect of loop nest depth on density and on per-prefetch slice cycle-time reduction. For each loop cluster, we suc-

Innermost loop function	Nesting	# stmts	Slice density	Cycle-time reduction
179.art				
train_match	4	290	86%	-2.5%
	3	288	44%	0.1%
	2	224	4%	0.0%
	1	120	3%	0.0%
train_match	2	224	5%	0.5%
	1	120	3%	0.0%
181.mcf				
refresh_potential	2	27	48%	7.9%
	1	22	18%	4.7%
primal_bea_mpp	3	487	71%	0.5%
	2	137	42%	4.2%
	1	88	21%	4.0%
183.equake				
smvp	3	358	10%	3%
	2	156	4.5%	0.8%
	1	155	2.5%	0.8%
188.ammp				
mm_fv_update_nonbon	4	993	26%	0.5%
	3	681	26%	0.2%
	2	121	9%	0.2%
	1	28	30%	0.0%
eval	3	27234	53%	-16%
	2	27178	0%	0.0%
	1	27152	0%	0.0%

Table 6.2: Slice densities for the different configurations of the most interesting clusters

```
while (1) {
    a = PICK_INT(1 , numcells);
    acellptr = carray[a];                                       (1)
    atileptr = acellptr->tileptr  ;                             (2)
    atermptr = atileptr->termsptr ;                             (3)

    for(t=atermptr; t; t=t->nextterm) {                         (4)
        ttermptr = t->termptr ;                                 (5)
        ...
    }
    ...
}
```

Figure 6.7: Loops with lots of dependent instructions have a small number of possible densities (300.twolf).

cessively strip the outermost loop, showing the density of the resulting slice and the speedup resulting from applying just this slice. Using this data, we exclude clusters that generate slices with a density greater than 60%. The 60% threshold is aggressive and permissive; it avoids ever dropping a favorable configuration. While it does retain some dubious clusters with extremely large slices that might simply add overhead at runtime, in practice we find that these candidates are eliminated by the compiler anyway because they make a library call the compiler cannot generate code for.

Prefetch-point selection: Having characterized the loop nest sizes and the space of clustering decisions, we now turn to the effect of prefetch-point selection on selected clusters. The majority of loop clusters have 1-4 prefetch point candidates with widely varying densities, and deciding about them is easy. We find that the loops with hundreds of prefetch point candidates break

Cluster	Nesting	DV	Prefetch points	Common densities
175.vpr I	2	31.2%	18	7.72%
175.vpr II	2	20.1%	47	10%, 23%, 34%
175.vpr II	1	3.2%	136	2.7%, 13.9%
179.art I	4	32.1%	7	100%
179.art II	2	19.7%	9	5%, 18.2%
179.art III	1	2.0%	19	9.5%, 23.8%
181.mcf I	2	51.3%	54	75.1%
181.mcf II	1	51.3%	52	72.3%
181.mcf III	1	16.4%	12	25%, 16.7%, 8.33%
183.equake I	3	67.7%	200	1.18%, 2.11%, 9.8%, 82.7%
183.equake II	2	62.7%	200	4.8%, 5.6%, 48.8%
188.ammp I	2	51.2%	32	0.2%, 0.1%
188.ammp II	1	45.6%	3	61.5%
188.ammp III	1	11.4%	3	1.25%
256.bzip2 I	1	31.9%	17	34.4%, 43.8%
300.twolf I	2	26.6%	7	8.3%, 9.2%, 32.4%, 93.5%
300.twolf II	2	16.8%	20	0.8%, 2.8%, 6.9%, 40.3%
300.twolf III	1	4.1%	30	0.6%, 2.9%
sphinx I	3	83.5%	472	83.5%, 48.3%
sphinx II	1	35.2%	10	5.8%, 76.1%
sphinx III	1	5.1%	6	35.7%

Table 6.3: Size of the prefetch-point selection state-space, with common densities for different prefetch-points. DV stands for dereference volume as defined in Section 6.2

down into a small number of nested equivalence classes because of the presence of low-ILP dependence chains. The presence of a loop-carried dependence ensures that including one of the dereferences in an equivalence class results in all the others being included. Figure 6.7 illustrates this pattern. Selecting any of the dereferences in statements 1–3 as the prefetch point will include all 3 statements in the backward slice. Density will thus remain the same. Selecting 4 or 5 would add both. Thus, there are only two legal densities in this loop nest, assuming no dereferences (ie. only computation) in the elided portions.

Table 6.3 enumerates some of the major loop clusters that test prefetch-point selection and the number of available prefetch point candidates — statements in the innermost loop of the cluster that contain pointer dereferences — for each. It also shows the most common densities for these loop clusters. In all but one of our applications, the largest possible density bounds the critical path to the last load as opposed to the computation performed using the loads in a loop cluster. Once again, our simple density threshold successfully picks a good prefetch point for all slices, while relying on overly large slices to be pruned during code-generation.

The notable exception to this pattern is 183.equake, where the presence of independent loads is common, causing multiple parallel dependence chains in a loop because of its multi-dimensional array data structures. Figure 6.8 illustrates this. Modifying the compiler to slice for multiple prefetch points per cluster allows us to explore the space of all possible combinations, but

```
for (i = 0; i < nodes; i++) {
  next = Aindex[i];
  sum0 = A[next][0][0]*v[i][0]  + A[next][0][1]*v[i][1]
             + A[next][0][2]*v[i][2];
  sum1 = A[next][1][0]*v[i][0]  + A[next][1][1]*v[i][1]
             + A[next][1][2]*v[i][2];
  sum2 = A[next][2][0]*v[i][0]  + A[next][2][1]*v[i][1]
             + A[next][2][2]*v[i][2];
  ...
}
```

Figure 6.8: 183.equake consists mostly of loops with multiple dependence chains.

once again we are either left with good density slices that fail to prefetch all important loads, or high density slices that are unable to run sufficiently far ahead of the prefetch thread. We prune the latter candidates from further consideration. A prefetch engine that can prefetch for multiple iterations in parallel — and so utilize all available prefetch bandwidth for independent iterations — may be able to consider such slices more aggressively.

Compiler back-end and prefetch kernel characteristics: Clusters and slices that fit the criteria of previous phases are now ready for code generation. Table 6.4 summarizes some characteristics of the resultant prefetch kernels in the TwoStep ISA for our applications — the number of individual kernels for each application, their total static size in instructions, and the number of registers utilized. We also present corresponding data from the manually-generated prefetch kernels of the previous chapter. As can be seen, the automatically generated kernels are less parsimonious than the hand-crafted versions along

Application	Kernels		Static size		# Registers		
	C	H	C	H	C	R	H
175.vpr	5	-	100	-	77	12	-
179.art	2	1	40	29	26	14	9
181.mcf	3	3	221	50	139	32	14
183.equake	1	-	29	-	30	12	-
188.ammp	4	-	514	-	355	48	-
256.bzip2	2	-	120	-	87	12	-
300.twolf	9	1	305	33	271	31	20
sphinx	9	2	935	99	671	31	16

Table 6.4: Vital statistics for the slices generated by our compiler (C), and comparisons with the hand-crafted slices from Chapter 5 (H).

each of these dimensions:

- The number of kernels goes up partly because the compiler is not smart enough to merge sibling clusters, and in a few cases because it generates kernels not covered in the hand-crafted case.

- The sizes of the prefetch kernels goes up because the compiler performs no peephole optimizations, resulting in redundant COPY and JUMP operations. We perform JUMP chaining to eliminate empty basic blocks in the prefetch program. However, we do not eliminate JUMPs to the next PC. We found that these peephole optimizations had no effect on prefetching effectiveness or cycle count; the bottleneck in executing our slices is memory and pull latency rather than the number of instructions executed in the L2.

- The TwoStep compiler currently performs no register allocation, always creating a new name rather than recycling free ones. The column R in Table 6.4 shows the true register requirements for our applications after straighforward manual register allocation. With the exception of one slice in 188.ammp, all our applications require 32 registers or fewer, even though the compiler remains oblivious to any register-capacity constraints at this time. In production it will need to be enhanced to occasionally spill.

Stitch code in the main program: Once the prefetch kernels are generated, the compiler must augment the main program for two reasons: inserting stitch code to trigger different prefetch kernels at stitch points, and inserting pulls at the start of loop iterations being prefetched for. Compared to manual kernels the overhead due to stitch instrumentation increases for two reasons: the increased fragmentation into prefetch kernels we alluded to above increases, and a conservative algorithm in the compiler that occasionally stitches variables that are never used by the prefetch program. These redundant variables also cause some increase in the register footprint of our prefetch kernels. They arise because our implementation maintains pointer-aware reaching definitions by statement rather than symbolic location in order to conserve compile-time space.

6.6 Discussion: TwoStep vs prior precomputation compilers

As detailed in Chapter 2, precomputation-based prefetching has been studied in several instances of prior work [54, 76, 77]. Most such studies have either computed slices in hardware or pursued post-compilation binary translation. Computing slices in hardware restricts the scope of individual slices, while binary translation detects only simple pointer-chasing patterns. Both these approaches are less effective at addressing the more complex interleavings of spatial and pointer access that we demonstrated in Chapter 3. The state of the art in thorough compiler-based precomputation is the work of Kim and Yeung [47]. We focus on this study in our comparison.

Kim and Yeung's compiler uses 2 kinds of profile information — loop iteration count profiles and cache miss profiles — to select compute precomputation slices for execution in spare hardware contexts of a simultaneous multithreading (SMT) processor. The compiler consists of three major phases: slice generation, prefetch conversion, and threading scheme selection. Slice generation consists of selecting stores to start the slice at using the cache miss profile, computing a slice back 2 loop nests. Once the slice is computed, prefetch conversion consists of removing stores and replacing loads with non-blocking variants. Finally, threading scheme selection considers two alternatives to simple serial preexecution — *doall* which speculatively updates the inductive variable for each iteration and runs later iterations speculatively in additional SMT threads; and *doacross* which performs a more detailed analysis of loop-carried dependences to decompose loop iterations into a ‘backbone’ and ‘ribs’, so that ribs may be executed in parallel.

This scheme — which we refer to as the SMT compiler — has much in common with our TwoStep compiler: a dependence on loop iteration count profiles, pointer analysis and slicing; the crucial decision of what to prefetch or what load to start backward slices at; sandboxing prefetch threads from making architecturally-visible changes. There are also several points of difference in approach:

1. The SMT and TwoStep compilers live in very different contexts in terms of hardware budget. The SMT compiler assumes a full processor ISA for prefetch threads with potentially multiple threads in flight. TwoStep consists of a simple controller that is little more than a state machine, leaving processor resources for other uses, and also simplifying our code generation.

2. Using a prefetch controller at the L2 is also more parsimonious than processor threads in terms of cache bandwidth. Since our prefetch controller sits at the L2 we only pay half the round-trip latency and bandwidth for each memory access. The reduced latency is especially important for sequential pointer-chasing.

3. The SMT compiler uses a simpler stitch-point selection criteria than we do — to simply stop two loop nests above the prefetch point. We explore

more aggressive possibilities and use the post-slicing density metric described above to backtrack and prune outer loops. Our more aggressive iterative solution shows 9% speedup for twolf as compared to the 2% they show, a difference which is significant given the extra hardware and multiple parallel prefetch threads of the SMT compiler.

4. Our pull instructions have about the same overhead as the semaphores in their implementation; however pulls are superior in two ways. First, the semaphores of the SMT compiler fix the application to a fixed number of *iterations*, where a FIFO-based approach measures the amount of potential pollution more precisely allowing us to be more aggressive in some cases. Second, using a FIFO decouples prefetch distance from pollution. Tighter loops can thus benefit from a larger FIFO and prefetch distance without risking pollution in the DL1.

5. The SMT compiler relies on cache-miss profiles generated using cache simulation. We use simple static models instead and rely for correction on backtracking in later phases. As a result, we are able to generate profiles using native rather than simulated execution. The time taken for cache simulation is proportional to the size of the dynamic execution of interest; the extra time taken by backtracking depends on application complexity. For applications in the SPEC suite, the two are comparable.

6. Having multiple prefetch threads in flight addresses a concern for TwoStep — sequential prefetch threads fail to use all available prefetch bandwidth.

This can become important in tight loops. As we show in Chapter 7, combining a precomputation-based scheme with a history-based scheme recovers a lot of the benefit in a simpler and more modular manner.

These differences are largely a result of the different hardware contexts of our respective studies. Given multiple parallel contexts, Kim and Yeung focus on ways to maximize their use, while TwoStep's design was driven by the desire to minimize the latency of pointer chasing. This latency is crucial in the patterns of serialized prefetching combining complex sequences of pointer-chasing and spatial offsets in some applications that we observed using DTrack. We now evaluate slices of the TwoStep compiler using several static metrics, deferring the more comprehensive evaluation of the toolchain to the next chapter.

6.7 Summary

This concludes our description of the TwoStep compiler. Our detailed surveys of the state space that the compiler must search serve to validate its density-based policies. We have shown that the state space, suitable decomposed, is not overly large and that a relatively simple compiler organization serves to find all opportunities in the form of favorable prefetch slices. The detailed analysis also uncovers the limitation of precomputation-based prefetching responsible for 179.art's lack of speedup: that certain kinds of loops with lots of dereferences per iteration organized in multiple dependence chains need a favorable compute-store ratio to be effectively prefetched. The 'tighter' the loop in terms of computation, the harder it is to effectively prefetch all of the different loads in the loop. Aside from this limitation, however, our compiler successfully handles a wide variety of applications and successfully converges on the right clustering and slicing decisions to compare very favorably with manually-generated kernels. While our manual versions have fewer static kernels, often combining multiple kernels where the compiler cannot, and unifying loops with identical access patterns, the compiler is able to obtain nearly all the speedup obtained manually.

The compiler performs whole-program analysis based on detailed pointer information. The more heavyweight analysis requires multiple context-sensitive traversals of an application's source code, one for each candidate slice processed during density measurement and code-generation. Compiling our largest codebases — sphinx — currently takes over 2 hours. Recent advances in adaptive on-demand context-sensitivity [95] could be used to optimize these traversals. In the rest of this dissertation, we focus on evaluating the resulting kernels, and on identifying the strengths and weaknesses of TwoStep.

Chapter 7

Evaluating TwoStep

Having described the TwoStep microarchitecture and compiler we now perform a detailed evaluation and characterization of TwoStep for our applications. Our results are broadly divided into two categories: comparison studies to measure the benefit of TwoStep relative to different approaches, and state-space explorations to better understand the strengths and weaknesses of TwoStep. In these results, we prune from consideration 4 applications with low memory usage that TwoStep fails to improve: 165.gzip, 177.mesa, 186.crafty, and 176.gcc. We were unable to compile 176.gcc and 197.parser because the TwoStep compiler runs out of memory.

We begin by measuring the coverage and accuracy of TwoStep prefetching for our applications using the methodology detailed in Section 3.3, showing that TwoStep successfully prefetches for a broad spectrum of access patterns. We then measure how this effectiveness with access patterns translates to aggregate speedups, comparing overall cycle-count reductions due to TwoStep with two prior prefetching approaches. Our results show that TwoStep's strengths are complementary to prior approaches; it especially performs well on extremely irregular applications such as sphinx, 188.ammp and 300.twolf that other techniques are unsuited to.

The next three sections delve into the reasons for these differing strengths. In brief, an application may be better suited to forward-looking *precomputation-based* prefetching or backward-looking *history-based* prefetching. History-based prefetching relies on finding patterns (usually spatial) in the dynamic address stream of an application. It is better suited to applications with spatial locality. Precomputation, on the other hand, can handle more complex access patterns where the address stream does not have a reliable pattern; however it requires a lot more sequential chaining between prefetches to generate accurate prefetches. As a result, it requires more computation per loop iteration to reliably provide improvements. We demonstrate this dichotomy first with a microbenchmark study, then with a more detailed characterization of prefetching in real-world applications to explore the relative strengths of region prefetching and TwoStep.

The final sections assess the relative importance of three important parameters of our system: DRAM latency, the capacity of TwoStep's FIFO, and the latency of pulls in transferring cache-lines from FIFO to DL1. These results support our choice of baseline and show that implementing TwoStep is a realistic proposition on current and future hardware.

7.1 The effectiveness of TwoStep prefetching

A prefetch technique is traditionally evaluated along two dimensions: by its accuracy, and by its coverage. TwoStep's *accuracy* is consistently high.

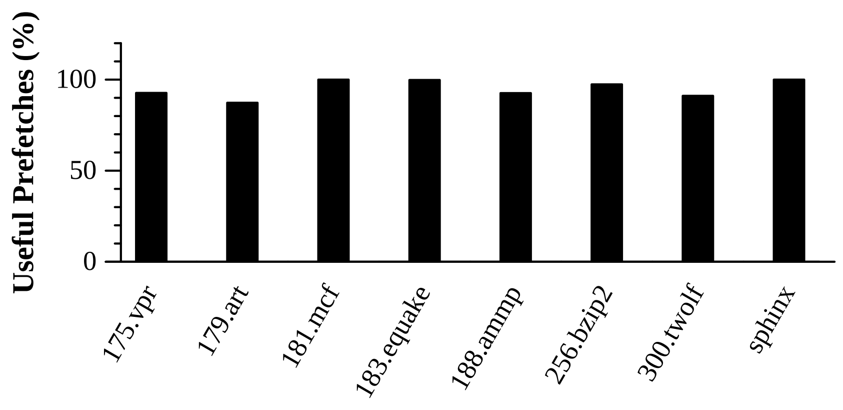


Figure 7.1: The accuracy of TwoStep prefetching for our applications.

We measure accuracy as the fraction of cache-lines prefetched into the DL1 that were used before eviction. Figure 7.1 shows that this fraction is uniformly high across all our applications; 179.art exhibits the worst accuracy of 87%. As a result of the high accuracy, TwoStep prefetching rarely increases an application's bandwidth requirements to main memory. Indeed, as Figure 7.2 shows, it sometimes reduces cache misses at the DL1 or the L2 as accurate prefetches improve temporal locality in the caches. We now describe this figure in more detail as we focus on the coverage of TwoStep.

Evaluating coverage: Figure 7.2 shows the misses remaining in the DL1 and the L2 after TwoStep prefetching relative to a baseline with no prefetching. Reductions in DL1 misses are due to useful pulls, and we return to these in more detail in the Section 7.6. We separate misses in the L2 to 3 separate categories: misses that were exclusively due to prefetches (i.e. miss latency

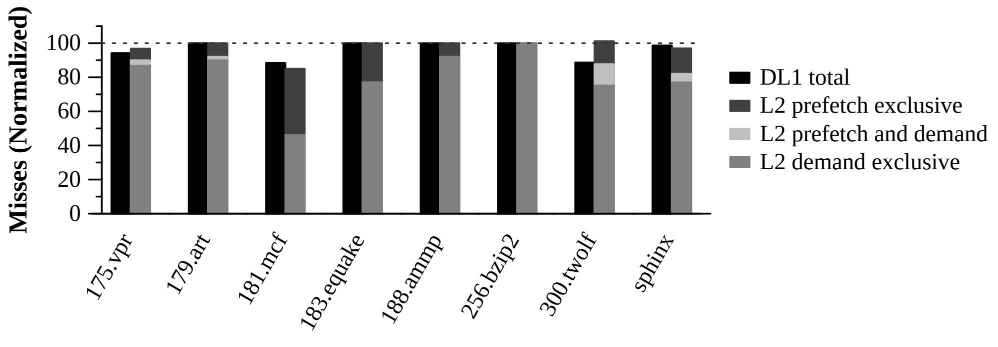


Figure 7.2: Aggregate misses remaining after TwoStep prefetching. This metric underestimates the improvement due to TwoStep.

was either entirely overlapped, or the prefetch was useless), misses that were initiated by prefetches but subsequently also by demand fetches (i.e. miss latency was partially overlapped by prefetching), and misses that were initiated exclusively by demand fetches (i.e. no latency was overlapped by prefetching). Figure 7.2 shows that while prefetches initiated by TwoStep are mostly accurate, different applications are able to leverage such prefetches to varying degrees. In 181.mcf, for example, TwoStep reduces total L2 misses by 17% and successfully overlaps all the latency of nearly half the remaining misses. In 256.bzip2, however, the compiler is unable to generate any prefetch programs with good densities, and so TwoStep provides no benefit.

Figure 7.2 exposes two disadvantages of using misses or miss-rate as a metric for measuring L2 coverage. First, the reduction in aggregate DL1 and L2 misses often underestimates speedups as we show later. Second, the ratio of demand misses to prefetch misses also serves as a poor indicator of speedups

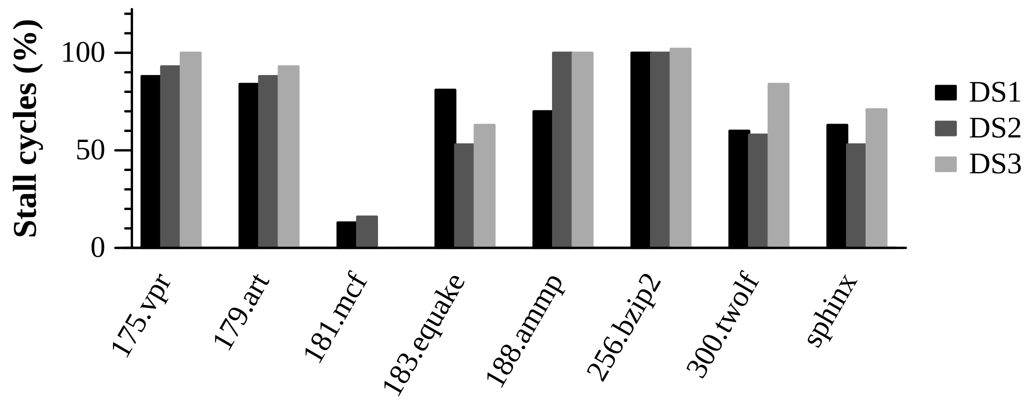


Figure 7.3: Stall cycles remaining after TwoStep prefetching for the most frequently missing data structures (DS1 and DS2 and DS3).

due to prefetching. Demand fetches and prefetches are often overlapped by the memory system in 3 applications: 188.ammp, 300.twolf, and sphinx.

These drawbacks have a single basic cause: pure miss counts are often poorly correlated with performance in modern systems because of the complexity of queueing and scheduling decisions between multiple misses in the memory hierarchy. Instead, our metric of choice is more immediate: time spent stalling due to memory latency. As described in Section 3.7, we track cycles that the pipeline commits no instructions, assigning blame to the data structure of the load at the head of the reorder buffer.

Figure 7.3 breaks down the effect of TwoStep prefetching on stall cycles for 3 major data structures in our applications. TwoStep consistently reduces stall cycles across a wide variety of benchmarks and access patterns. The greatest reductions occur in memory intensive applications with irregular

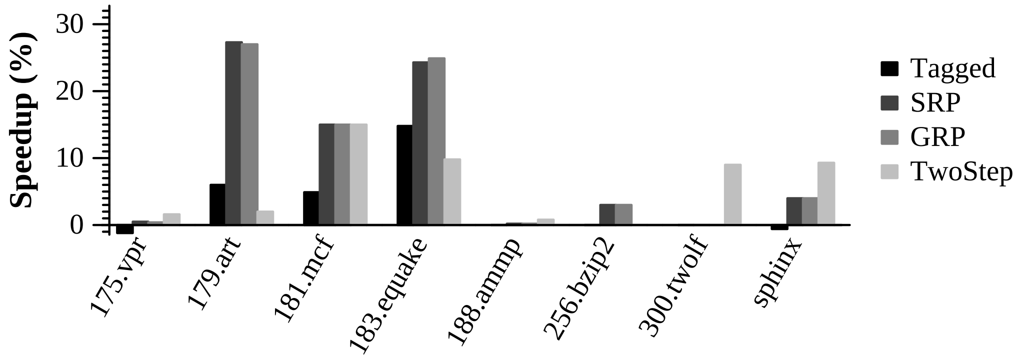


Figure 7.4: Comparing prefetch techniques

access patterns — 181.mcf, 300.twolf and sphinx. Lower-magnitude reductions can be seen for 183.equake (regular but memory-intensive) and 188.ammp (irregular but with lower cache miss-rates). The combination of the reductions in stall cycles and the fraction of useful prefetches shows that TwoStep is successful in its core design goal: prefetching a wide variety of access patterns. It also serves to highlight the applications where we do better than others. We explore this question further in the next section.

7.2 Comparing prior approaches

As described in Chapter 5, Figure 7.4 shows the aggregate speedups of TwoStep for our applications relative to a baseline with no prefetching. We also compare TwoStep with one short-range and two prior long-range prefetching techniques — *Tagged Prefetching*, *Scheduled Region Prefetching* (SRP) [55] and Guided Region Prefetching (GRP) [98], respectively. Tagged prefetch is an example of a common category of simple hardware prefetching included in many production microprocessors. It prefetches the next cache-line on an L2 cache miss, marks cache-lines so prefetches using an extra bit, and continues to prefetch cache-lines and set their bits on the first use of a prefetched cache-line. This approach allows limited lookahead and concomitant improvement for simple spatial patterns, but fails to improve less regular applications. Our results confirm this.

SRP uses the L2 prefetch controller to trigger spatial prefetches in an aligned 4KB region on encountering L2 misses, taking care to prioritize demand fetches and prefetches of different regions and bounding the pollution in the L2 due to useless regions when the application has no spatial locality. GRP is a descendant of SRP that performs aggressive compiler analysis to augment important loads in the application with prefetch hints. The prefetch controller in GRP also performs content-based pointer prefetching that allows it to run ahead of the application by a statically bounded number of iterations. In spite of its support for various kinds of pointer-based prefetching, GRP's results are similar to those of SRP, getting most benefit from spatial access patterns but with greatly improved prefetch accuracy and greatly reduced memory traffic relative to SRP. All three sets of results use a common Rambus model. However, GRP uses different compiler and simulator infrastructure and is therefore measured against its own baseline.

TwoStep outperforms GRP and SRP on the 4 most irregular applications: 300.twolf, sphinx, 175.vpr and 188.ammp. Speedups are bounded by the memory intensiveness of the application; 175.vpr and 188.ammp have fairly low miss-rates. Another memory-intensive application with irregular access patterns is 181.mcf, and TwoStep provides significant speedups that are nearly identical to the prior techniques. However, SRP and GRP improve 181.mcf only due to accidental spatial locality in its layout; allocation and access follow the same path through the data structure. We believe the use of 181.mcf's simplex algorithm in a more general graph-optimization application with multiple possible paths of access would not attain this level of spatial locality, making TwoStep more effective in comparison.

Figure 7.4 also highlights the areas where TwoStep is not as effective as prior approaches. 179.art and 183.equake are regular applications that SRP and GRP are able to significantly speed up. TwoStep also shows speedups for them, but the speedups are not as significant. This lack of improvement arises because the precomputation approach forces TwoStep to serialize prefetches where approaches tuned for just spatial locality can issue multiple prefetches in parallel taking advantage of all available prefetch bandwidth. The effect of this parallel bandwidth depends on the relative quantities of computation per memory access in an application; thus the difference is widest for 179.art which spends nearly 90% of its time in extremely tight loops with 2-6 instructions of computation per memory access. 183.equake has more computation per memory access, concomitantly improving the effectiveness of TwoStep. We now support this reasoning in a microbenchmark study.

```
class Object:               // Size: one cache-line
    Object* next[4]
    int x[4]                // Padding

Object f[OBJECTS]           // Size: 10x L2 capacity
                            // Each element's next pointers
                            // initialized randomly.
Object* currObj = f

Archetype (regularity, computation):
    do 100-regularity times:
        do computation times:
            sum = (sum + currObj->value)%8
        currObj = currObj->next[i%4]

    do regularity times:
        do computation times:
            sum = (sum + currObj->value)%8
        ++currObj
```

Figure 7.5: The `Archetype` microbenchmark for exploring the application coverage of different prefetch schemes

7.3 Microbenchmark study: The space of application behavior

This section presents our coverage study to show that TwoStep is more broadly-applicable than prior approaches. We describe a simple microbenchmark that allows us to tune two significant application features - the compute vs memory-access ratio (`computation`) and the fraction of regular vs irregular and hard-to-predict memory accesses in the dynamic address stream seen by the memory hierarchy (`regularity`). We design our microbenchmark to

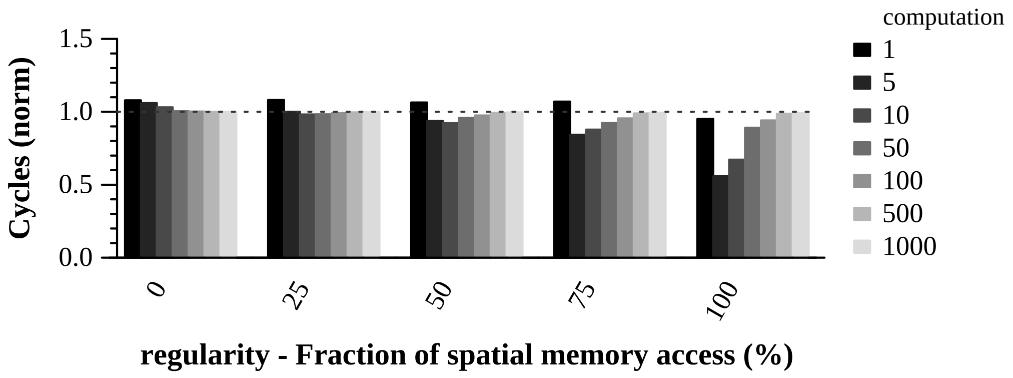


Figure 7.6: The coverage of GRP by iterations of `computation` per object accessed, and by `regularity` (both variables from Figure 7.5). GRP is biased towards the regular side of the space.

exaggerate the contrast between extremely regular and extremely irregular access. Figure 7.5 shows the basic structure of our *Archetype* microbenchmark. Archetype consists of a large array of objects an order of magnitude larger than L2 capacity and a series of traversals over it of variable regularity. To eliminate intra-object misses, each object in the array is aligned and sized to fit exactly in an L1/L2 cache-line. Each object contains pointers that are initialized to point to four other objects in the array chosen at random. Each iteration/call to Archetype now traverses the array in a combination of first completely irregular pointer-based access and then completely regular stride-1 access with good spatial locality. Rather than try to enumerate the space of possible access patterns and object sizes we select these two types of access with extreme cache behavior and study the effect of their relative weight on different types of prefetching.

Given the Archetype microbenchmark we can now explore the speedups yielded by different prefetch schemes for different values of `regularity` and `computation`. These speedups may be summarized in the form illustrated in Figure 7.6. This graph shows speedups for 5 groups of bars corresponding to different values of `regularity` on the x-axis, so that the set of bars at 100 have perfectly spatial access patterns while those at 0 have no spatial access. Within each group of bars we vary `computation`, the amount of computation per memory access.

Figure 7.6 exhibits several distinct regions. First, areas with low values of `computation` per object (left-most bars in each group) present little opportunity for overlapping latency and GRP (not unlike other prior schemes) fails to provide speedup. Second, as we increase `computation` to extremely high levels (right-most bars in each group), Archetype enters the space of compute-bound applications. Again, speedup due to prefetching is limited in this case. Between these two extremes lie the range of values for `computation` where prefetching can potentially provide speedups. SRP and GRP only improve the regular side of this space, gradually decreasing speedups as Archetype accesses memory more irregularly in the groups on the left. This result is in agreement with findings of the original study; limitations in prioritizing between pointers and a hard limit on the slack available to the prefetcher are the major bottlenecks in improving irregular applications. The major improvement of GRP over SRP is reduced memory traffic due to compiler hints that suppress useless region prefetches.

Unlike SRP and GRP, TwoStep (Figure 7.7) improves both the regular

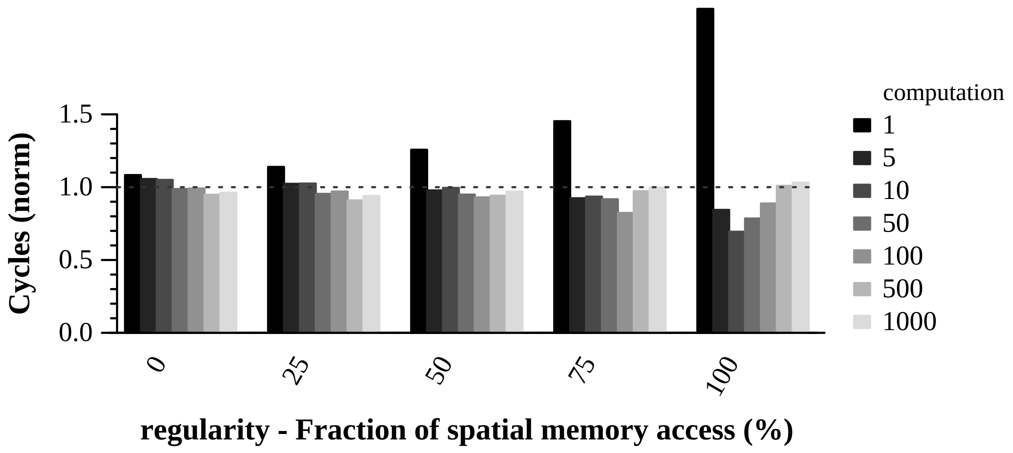


Figure 7.7: The coverage of TwoStep by iterations of `computation` per object accessed, and by `regularity` (both variables from Figure 7.5). Both regular and irregular applications now benefit from prefetching.

and irregular sides of the space, so that each group of bars shows significant speedups for some level of computation. There are two important secondary effects. First, the serialization of precomputation causes TwoStep to need more `computation` per memory-access to show speedups. We disabled our compiler's density checks to force it to prefetch at all levels of computation, and this causes significant slowdowns for tight loops. In practice our compiler simply excludes such loops from TwoStep prefetching. Comparing the regular side of Figures 7.6 and 7.7 also shows this effect – at 100% regular access, GRP shows the most speedup at a lower level of `computation` than TwoStep does.

Second, greater breadth in the application space is offset by degradation at some individual points in the space relative to SRP and GRP. As the speedup distribution graphs show, SRP and GRP usually have 1-2 points in the space with substantially higher speedup than TwoStep can manage.

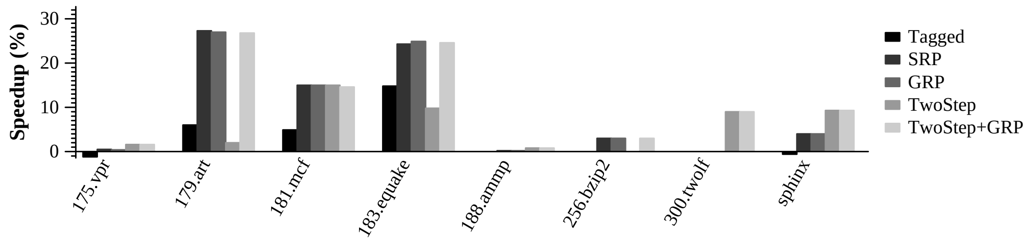


Figure 7.8: Combining SRP with TwoStep gives the best of both worlds.

Summary: In this section, we presented a novel method to study the coverage of a prefetch scheme in the space of applications. Varying application behavior rather than parameters of the system it runs on is a relatively understudied technique for highlighting the advantages and constituencies of different schemes. Our results show that TwoStep provides substantially greater breadth in the types of applications it can improve, at the cost of reduced speedups in the portion of the space that prior approaches have traditionally targetted. They also highlight the complementary strengths and weaknesses of history- and precomputation-based prefetching approaches: the former exploits prefetch bandwidth but requires address regularity; the latter exploits complex access patterns but requires more computation per memory access.

7.4 Combining history- and precomputation-based prefetching

The insight that history- and precomputation-based prefetching are complementary raises the possibility of combining them to get the best of both worlds. To explore this possibility we enhance the L2 prefetch controller to perform strided region prefetching when the TwoStep precomputation engine is disabled. Just like in SRP, region prefetches are scheduled with lower priority than demand fetches or the more accurate TwoStep prefetches, and are prefetched into the LRU way of the L2 cache without being pushed onto the FIFO. Figure 7.8 summarizes our results, extending the comparison in Section 7.2 with a new bar for our combined prefetching approach. As this Figure shows, combining TwoStep with region prefetching gives us the best of both worlds, providing the accuracy of precomputation-based prefetching in the extremely irregular applications that require it, and providing the bandwidth utilization of region prefetching in loops too dense for the TwoStep compiler to precompute for, and also in the rare cases of the prefetch thread falling behind the main thread and giving up in regular applications.

Using region prefetching without the compiler hints of GRP causes increased bandwidth requirements just like SRP. In principle it should be possible to add GRP's compiler analyses and hints to the TwoStep compiler, though they are currently implemented in separate compiler frameworks (Scale and C-Breeze, respectively). Combining spatial prefetch with TwoStep requires good pollution control and prioritization to manage low spatial prefetch accuracy. This is confirmed by experiments combining tagged prefetch with TwoStep, which show significant conflict between the two approaches and no speedups for irregular applications.

Having completed our comparison and synthesis of precomputation- and history-based prefetching approaches, we now conclude our evaluation

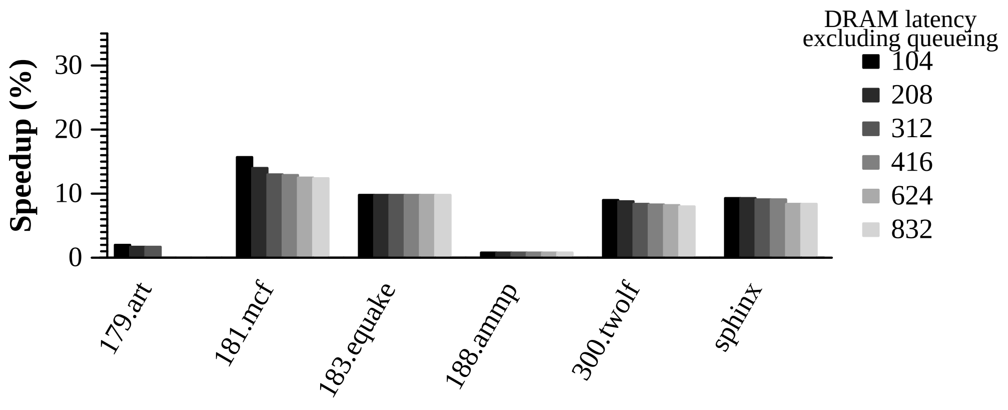


Figure 7.9: How TwoStep's speedups scale with growing memory latency.

with a series of sensitivity studies to study the effect of different system parameters on TwoStep's performance.

7.5 Effect of main-memory latency on prefetch effectiveness

An important question when studying speedups due to prefetching is how these speedups change as we increase latency to main memory. Figure 7.9 answers this question. For each application, the left-most bar shows the baseline RDRAM model used in the rest of this thesis, with RDRAM clocked at a cycle ratio of 4 relative to processor frequency. We model increasing latencies to main memory by changing just this cycle ratio without adjusting the relative times spent by each DRAM access in its different constituent phases: precharge, activation, the read/write itself, and queuing delay.

Figure 7.9 shows that increasing main-memory latencies reduces the

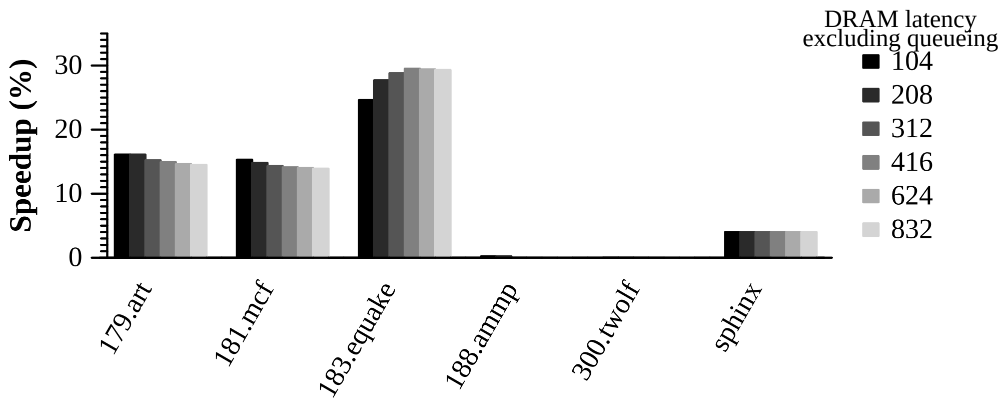


Figure 7.10: How GRP's speedups scale with growing memory latency.

speedup due to prefetching very slightly. For example, 300.twolf's speedup goes from 9.0% to 8.0% over a factor of 8 increase in RDRAM latency (average read latency increases from 92.5 to 742 cycles). Over the same space IPC drops by a factor of 4 from 0.66 to 0.15. This may seem implausible at first; as DRAM latencies grow we would expect the processor to be able to overlap less and less of the large latency by prefetching. To explain why this is not the case, we focus on the dependence structure of our programs. As DRAM latency increases, it becomes the primary factor deciding IPC. Since the dependence chains in an application are constant as DRAM latencies grow, the number of instructions that can execute overlapping with each dynamic DRAM access will tend to stay constant. Similarly, any prefetches issued will start at approximately the same instruction. Since main memory bandwidth is likely to be relatively highly utilized, the limited lookahead window in the out-of-order processor means that as we increase memory latency the ratio of RDRAM accesses to

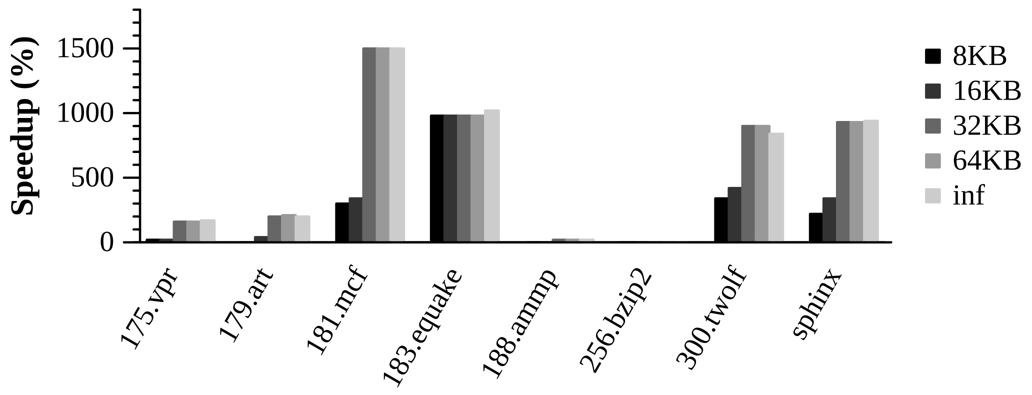


Figure 7.11: Sensitivity of TwoStep's speedups to FIFO capacity.

instructions committed remains the same. As a result the speedup due to prefetching is also largely maintained.

Figure 7.10 shows the corresponding figure for GRP rather than TwoStep; once again increasing memory latency has only a slight effect on prefetching effectiveness. In the case of 183.equake it even causes speedup to increase slightly up to a cycle ratio of 16 before tapering off. This is explained by the relatively high fraction of unutilized memory bandwidth for 183.equake at our baseline DRAM latency. As a result, it requires DRAM latencies to grow by a factor of 4 before memory bandwidth is nearly fully utilized. At that point the sequentialization between memory accesses kicks in as described above, and speedups stay largely constant past that point. Speedups due to precomputation-based prefetching are more likely to have larger dropoffs with increasing memory latency because of the increased sequentialization of accesses.

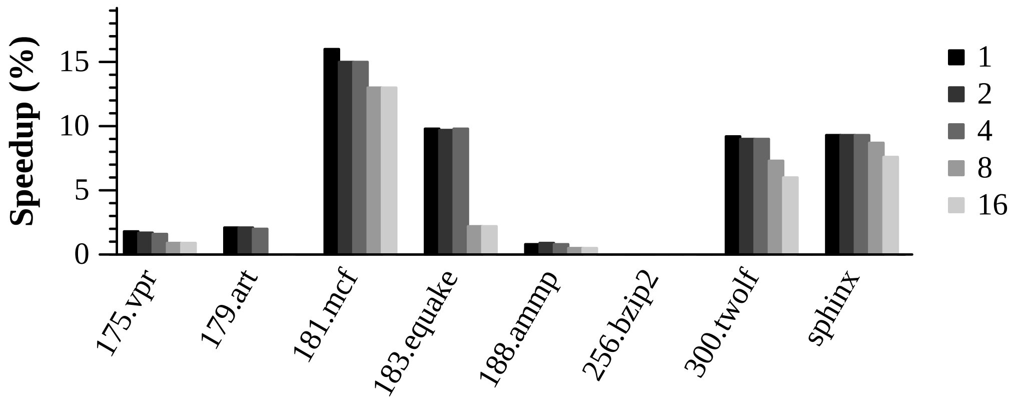


Figure 7.12: Sensitivity of TwoStep's speedups to latency of the first cache-line on a pull. Subsequent cache-lines arrive 1 cycle apart. Further increases in latency do not cause more dropoff; the rightmost bar for each group measures the speedup due to prefetching to L2 rather than DL1.

7.6 Sensitivity studies

The TwoStep design has two major parameters - FIFO capacity and pull latency - that must be realistic in order for it to be feasible. We now evaluate its sensitivity to these parameters. Figure 7.11 summarizes the speedups obtained by TwoStep for our applications and the sensitivity of these improvements to FIFO capacity. A 2KB (32-entry) FIFO suffices to provide most of the benefit of an infinite-capacity FIFO, indicating the effectiveness of the FIFO at realistic capacities for current technologies [2].

Figure 7.12 shows the effect of pull latency on TwoStep's speedups. We vary the latency of transfer of the first cache-line from FIFO to DL1, assuming pipelining allows subsequent cache-lines for each pull to arrive 1 cycle apart at the DL1. We find that across all our applications a 4-cycle pull latency gives us the same speedups as a 1-cycle latency.

Increasing the latency to 16 cycles or more causes demand fetches to hit in the L2 before the pull arrives in the DL1. The right-most bar in Figure 7.12 is thus a good indication of the relative benefit of TwoStep prefetching to the L2 and DL1 for our applications. Different applications benefit from prefetching to the DL1 to varying degrees, with memory-intensive applications like 181.mcf and 300.twolf getting most of their benefit from accurate prefetch to the L2, while regular applications like 179.art and 183.equake also benefit significantly from the pulls to the DL1.

7.7 Summary

This chapter presented a detailed evaluation of the entire TwoStep microarchitecture and compiler toolchain described in the previous two chapters. We have shown that TwoStep prefetching provides cycle-time reductions across all the applications we evaluated on relative to a baseline with no prefetching. Analyzing the results further, we find uniformly substantial accuracies, but wide variance in prefetch coverage, especially for tight loops and regular programs. While irregular applications are uniformly improved relative to GRP, regular applications often do significantly better with prior approaches. We explore why and show that the need to serialize dependent prefetches is a disadvantage for TwoStep when running such applications. More generally, precomputation- and history-based prefetching are complementary approaches and we identify the precise application characteristics that determine application affinity to one or the other.

Chapter 8

Conclusions

Prefetching is an attractive solution to growing memory latencies. Unfortunately, implementing prefetching well has been a challenge for modern systems researchers, largely because of the wide variety of application behavior seen by modern computer systems. Every prefetching system must make decisions on what to prefetch, when to prefetch it, and where to prefetch it to. It must make a high volume of these decisions without adding too much overhead. In this study we have highlighted the subtleties in making these decisions and the many ways that a mechanism that improves one decision for one set of applications may degrade the quality of another decision for a different set. One major such tension is between history- and precomputation-based approaches for deciding what to prefetch. Using past history utilizes prefetch bandwidth more efficiently and makes timing decisions easier, but may yield low-accuracy prefetches for complex irregular applications. Using precomputation guarantees accurate prefetches, but serial dependences between prefetches worsen the problem of timing prefetches. In this dissertation we addressed these interacting problems.

8.1 Summary of contributions

The major theme in this dissertation has been that the chaotic behavior of large applications is an artifact of insufficient analysis, and can be decomposed into more regularly-behaved components. We began by decomposing the address streams of applications by data structure and phase, and by showing that this process can give insight into each application's behavior and yield a *symbolic* access pattern for the major loops in an application. As applications grow more complex, general-purpose processors must be increasingly proactive in adapting to their changing needs over time. Data structures and loops are ideal high-level structures for designers to focus on in order to gain insight.

DTrack, our tool for data structure decomposition, highlighted the wide variety of behaviors in modern applications. Of the 8 applications we studied, 5 contribute 90% of their cache misses in just three data structures, while the other 3 can take as many as 100 data structures. While the phase transitions in our applications occur at the same points across all data structures, the behavior of different data structures and phases is widely variable. Our applications benefit from an application-specific sampling period at which to perform phase analysis. Combining phase and data structure profiles yields distilled summaries of the dominant access patterns in our applications, and highlights the first access to an object in a loop iteration as the most frequent cause of cache misses. Loop iteration footprints are tiny relative to cache capacities, allowing us to aggressively tune for these first object accesses.

We then used our understanding of these major loops to understand the drawbacks of prior prefetch approaches, and to design a prefetch scheme that addresses these drawbacks by orchestrating cache-lines into the level-1 data (DL1) cache in units of a loop iteration. TwoStep leverages modern compiler techniques to provide the memory hierarchy with a distilled picture of the application's access patterns. Prefetches originate in the level-2 (L2) cache to minimize address traffic and latency between dependent prefetches. Decisions of what to prefetch next are decoupled from when to prefetch. A FIFO between L2 and DL1 provides both a low-overhead flow-control mechanism that allows the rest of the system to largely ignore the possibility of pollution, and also prefetches data to the DL1 right before its use. We find these mechanisms to work harmoniously together.

The goal had been for this dissertation to provide a single set of mechanisms that are effective for the large variety of access patterns seen in the wild. From that perspective our results have been mixed. TwoStep works well for programs with irregular access patterns and reasonable levels of computation per memory access. While these criteria seem reasonable, finding benchmarks that fit them has been difficult, especially when coupled with toolchain-imposed constraints — we require C sources and our compiler overheads precluded running 3 SPEC2000 benchmarks. While we successfully improve irregular programs over prior work, our improvements for regular applications are lower than competing approaches. Understanding why this is so is one of the contributions of this dissertation: precomputation imposes an ordering on prefetches and so is unable to fully utilize available prefetch bandwidth. Rather than a technique that subsumes prior approaches, we have ended up with an understanding of the complementary strengths of our approach and prior techniques.

Prefetching can either look back at past history or look forward by precomputing an application's future requirements. We have quantified the complementary advantages of these techniques into two application-level properties. Applications with a low compute-access ratio can benefit from history-based prefetching *if* their access pattern is not too irregular. Applications with irregular access patterns are likely to require precomputation-based prefetching, *as long as* their compute-access ratio is not too low. If the reader remembers one fact from this dissertation, we recommend this one.

TwoStep is an elaborate system requiring profiling, whole-program analysis, ISA modifications and microarchitectural changes. Over the benchmarks we evaluated TwoStep over, the average improvement relative to prior approaches like SRP is insufficient to justify including the additional complexity of TwoStep in a production design. However, I believe future trends will make TwoStep more broadly applicable. As computers have become cheaper and more accessible the trend in the last 30 years has been for applications to grow more diverse (with new categories like streaming media and personal productivity), more complex (word processors check grammar and also perform speech recognition and synthesis) and more memory-intensive. These trends are likely to continue in future: the number of applications running concurrently on a system, the variety of applications, and the variety of phase behaviors in an application are all likely to increase. Applications that stream media but perform non-trivial computations in each iteration, such as speech recognition's beam search, are prime candidates for precomputation-based prefetching.

8.2 The roads not taken: Challenges for future work

When starting out, my goal was to explore ways in which the hardware-software stack could be designed to be more responsive to the needs of individual applications, and to determine the effectiveness of this approach in reducing the time taken to run different types of applications. Implications of this approach are that both hardware and software may need to change, and that the interface between the two could benefit from greater richness. In the process of writing this dissertation I have made many choices of avenues to pursue. While we have used the insights yielded by DTrack to improve prefetching, there are many alternative applications to these insights along three broad areas: improving static application layout, improving cache replacement, and improving scheduling of data movement into the caches.

Improving data layout: An application's data layout can be improved in two ways: either by improving heap allocators or by providing multiple address mappings for individual memory locations like the Impulse memory controller [14]. One interesting approach to improve an application's data layout is to provide not one version of malloc but multiple versions tuned for different types of access patterns, relying on compiler support to replace calls to `malloc()` in the application with an appropriate specialization. The most similar study to this in the literature is by Wilson et al. [101]. This approach is however limited to applications that rarely update their data structures; applications that update their data structures at even a low rate end up with a random data layout if they run long enough. Applications without updates to the dominant data structures will benefit from this approach; our compiler implementation shows, in combination with previous work, that determining access patterns statically is feasible. The open problem is translating access patterns into a taxonomy of allocation policies. We didn't have access to a broad enough range of applications to attempt such a taxonomy. Static allocation policies to address the most frequent access patterns are also synergistic with multiple address mappings to take less frequent access patterns into account.

Improving cache replacement: The second category of optimizations consists of ways to improve cache replacement. Cache replacement can be improved either by more adaptive policies [74, 81] or by more sophisticated cache partitioning. While both approaches have been tried in the past, a promising line of attack in either category is to explore in this context the potential of an online system to associate data structure categories with individual memory addresses. Creating a more coarse-grained form of DTrack analysis that can be performed online with low overhead could help improve cache bypassing and dead-block prediction decisions for either performance improvement or power reduction. A potential further refinement is to bind specific cache partitions to sets of data structures. Especially in combination with reconfigurable caches, this approach may help avoid conflict between data structures.

Improvements to prefetching: TwoStep prefetching can be improved in several ways. We outline three major ideas. First, TwoStep has lower speedups than region prefetching for extremely regular applications. We have shown that TwoStep and SRP can be combined without conflict to get the best of both worlds. This solution however suffers from the potential low accuracy and increased bandwidth requirements of SRP. Combination with GRP has been shown to be feasible, but compiler support for such a combination remains to be implemented. A second way to address regular/spatial applications is to use multiple prefetch threads like Kim and Yeung [47]. In the context of TwoStep, this will require the compiler to generate multiple versions for each prefetch kernel: one to perform the in-order pushes to the FIFO from the L2, and another in potentially multiple instances to run ahead and prefetch multiple iterations of a loop in parallel. Third, the TwoStep compiler currently emits extremely unoptimized code to run on the L2 controller. While our applications have shown no benefit from optimizing further, it is possible that new applications will be able to tolerate lower ratios of computation per memory access with more optimized prefetch kernels. Each of these is — in descending order of promise — a potential source of future improvement to prefetching for irregular and regular programs alike.

Nonetheless, this dissertation has articulated a new approach: of exploring the feasibility of dynamic adaptation using a richer interface between hardware and software, and of using dynamic adaptation to address more complex applications than have heretofore been taken into consideration in system design. While the implementation can be improved, fine-grained orchestration and data cache management is a valid and complementary approach to prior approaches that maximize prefetch bandwidth utilization.

Appendix

The figures in the following pages show, for each of the major data structures in our applications, the raw time-varying data every 50 million cycles for DL1 accesses, DL1 misses (L2 accesses), and L2 misses. We provide an overview of these figures, enumerating for each application the dominant data structures in terms of total cache misses and their access pattern. For more data on these data structures, consult Tables 3.3–3.5.

Benchmark	Data structure	Access pattern
164.gzip	`window`	Regular
	`prev`	Regular
	`inbuf`	Regular
	`fd`	Regular
175.vpr	`node`	Regular
	`heap`	Irregular
	`node_route_inf`	Irregular
	`linked_f_ptr`	Irregular
177.mesa	`Image`	Regular
	`Depth`	Regular
	`Vertex`	Regular
	`Normal`	Regular
179.art	`f1_layer`	Regular
	`tds`	Regular
	`bus`	Regular
181.mcf	`nodes`	Irregular
	`arcs`	Irregular
	`perm`	Regular
	`basket`	Regular
183.equake	`K[][]`	Regular
	`disp[]`	Regular
	`K[]`	Regular
	`K`	Regular
188.ammp	`atom`	Irregular
	`nodelist`	Regular
	`atomlist`	Regular
	`vector`	Regular
256.bzip2	`block`	Irregular
	`quadrant`	Irregular
	`zptr`	Irregular
300.twolf	`netarray[]`→`netptr`	Irregular
	`tmp_rows[]`	Irregular
	`rows[]`	Irregular

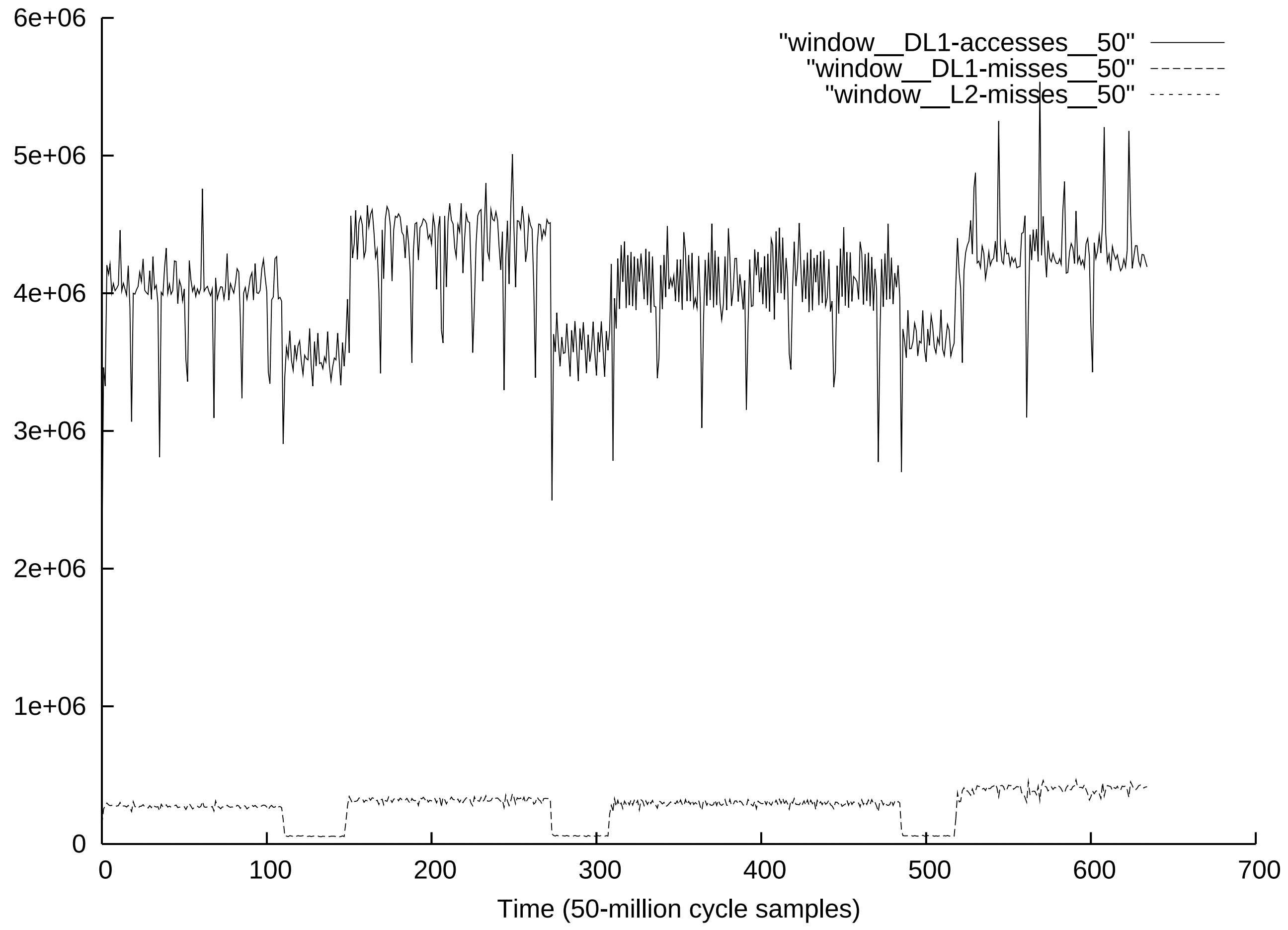


Figure 1: 164.gzip – window

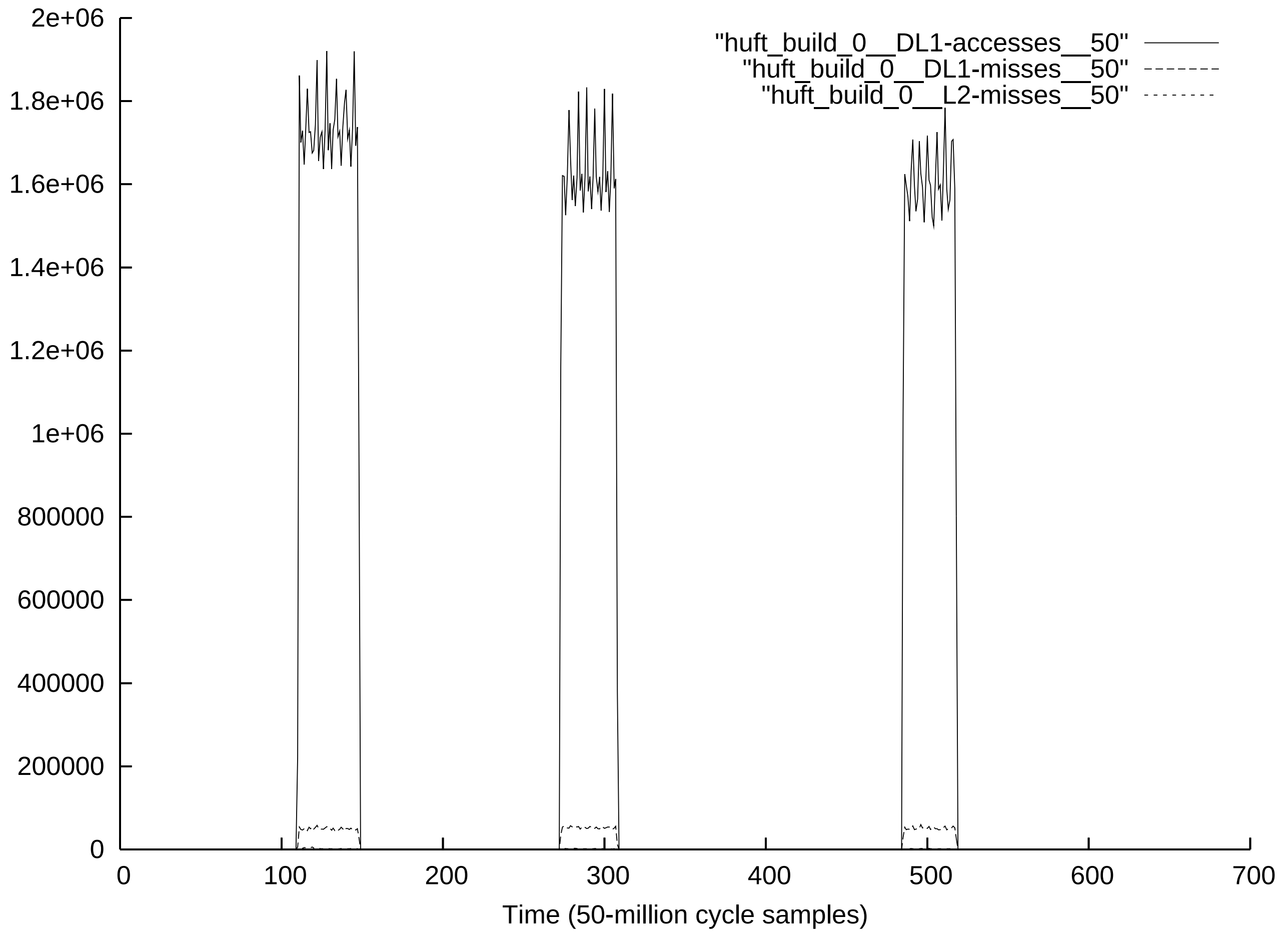


Figure 2: 164.gzip – prev

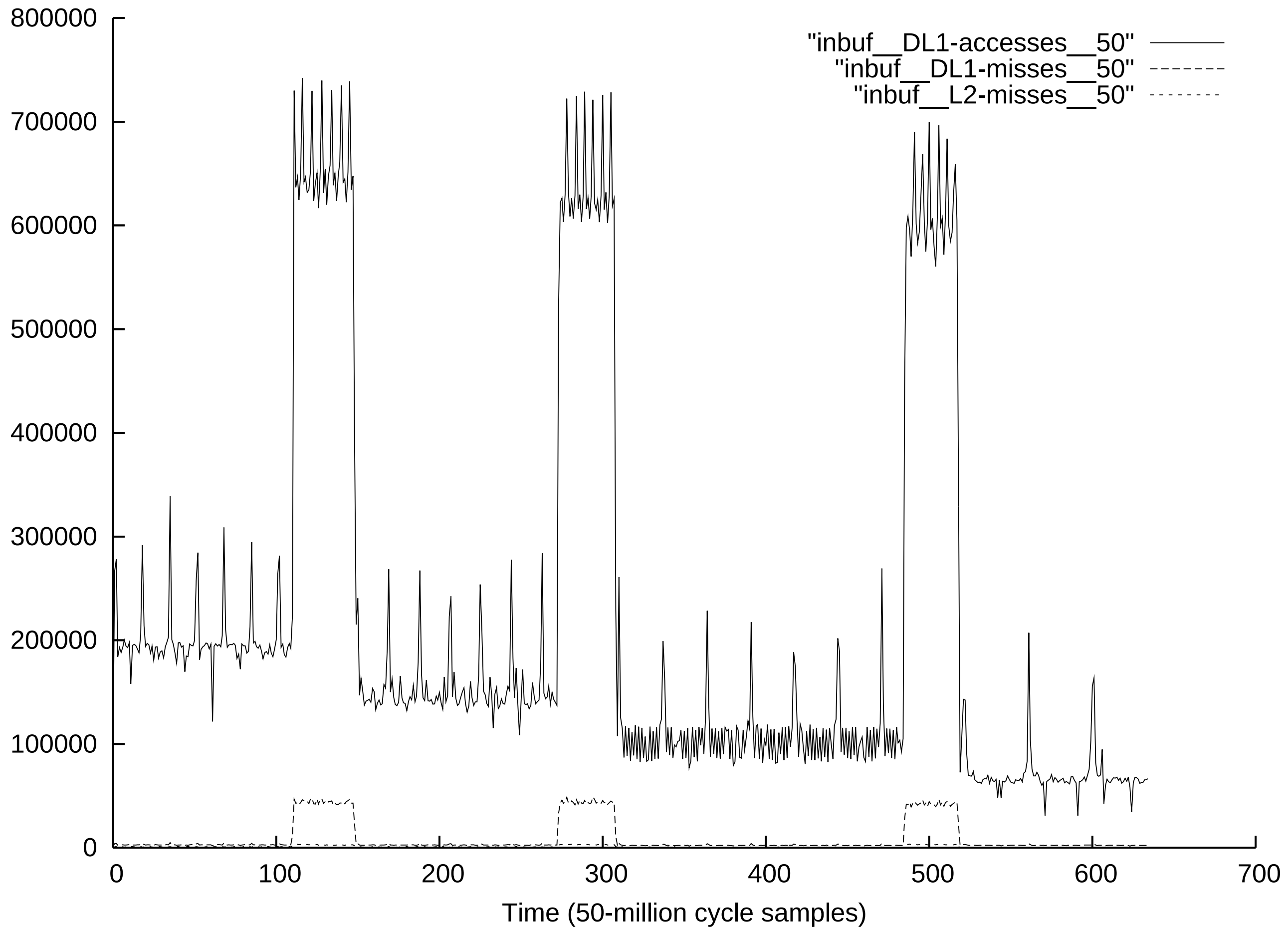


Figure 3: 164.gzip – inbuf

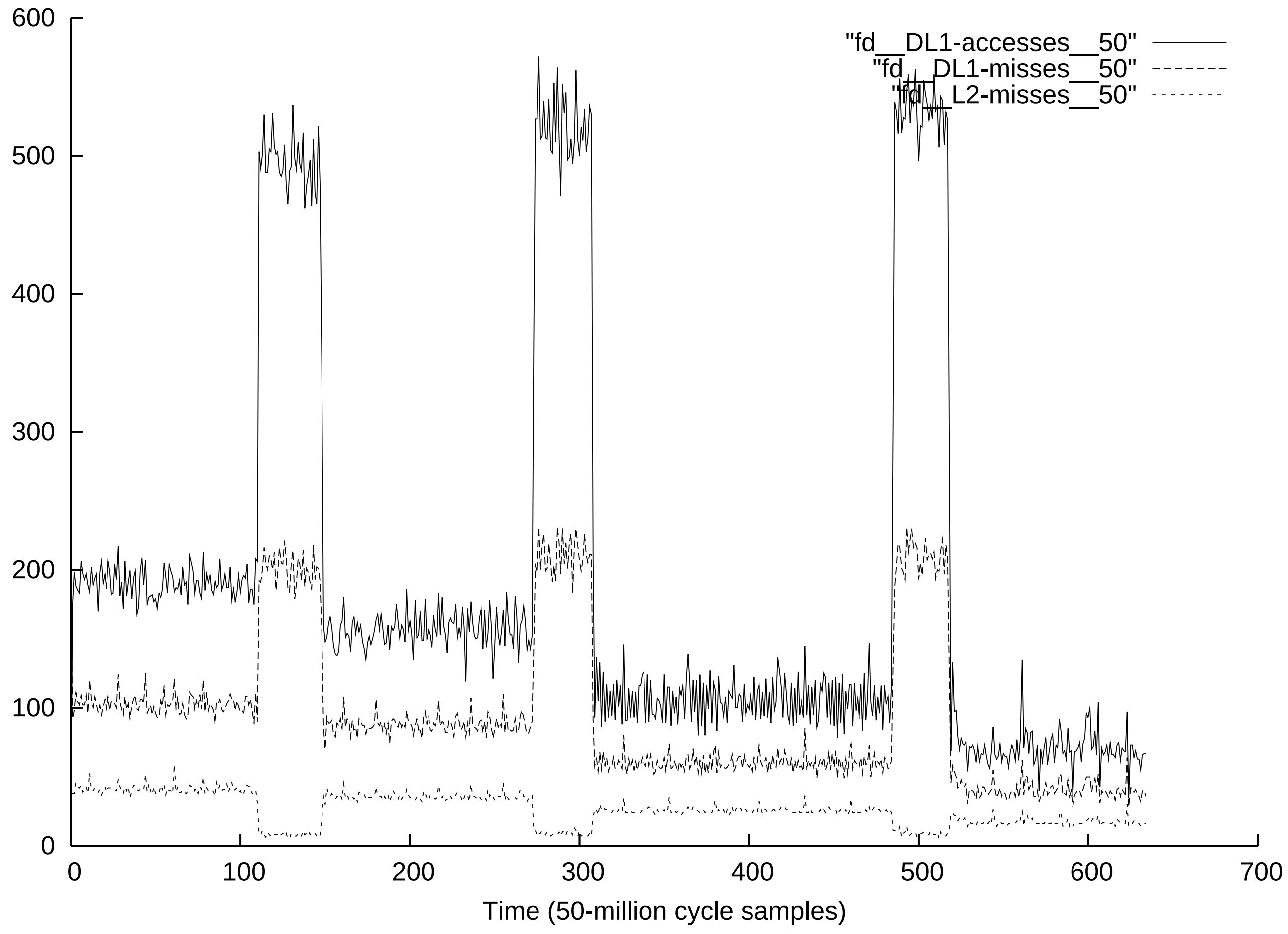


Figure 4: 164.gzip – fd

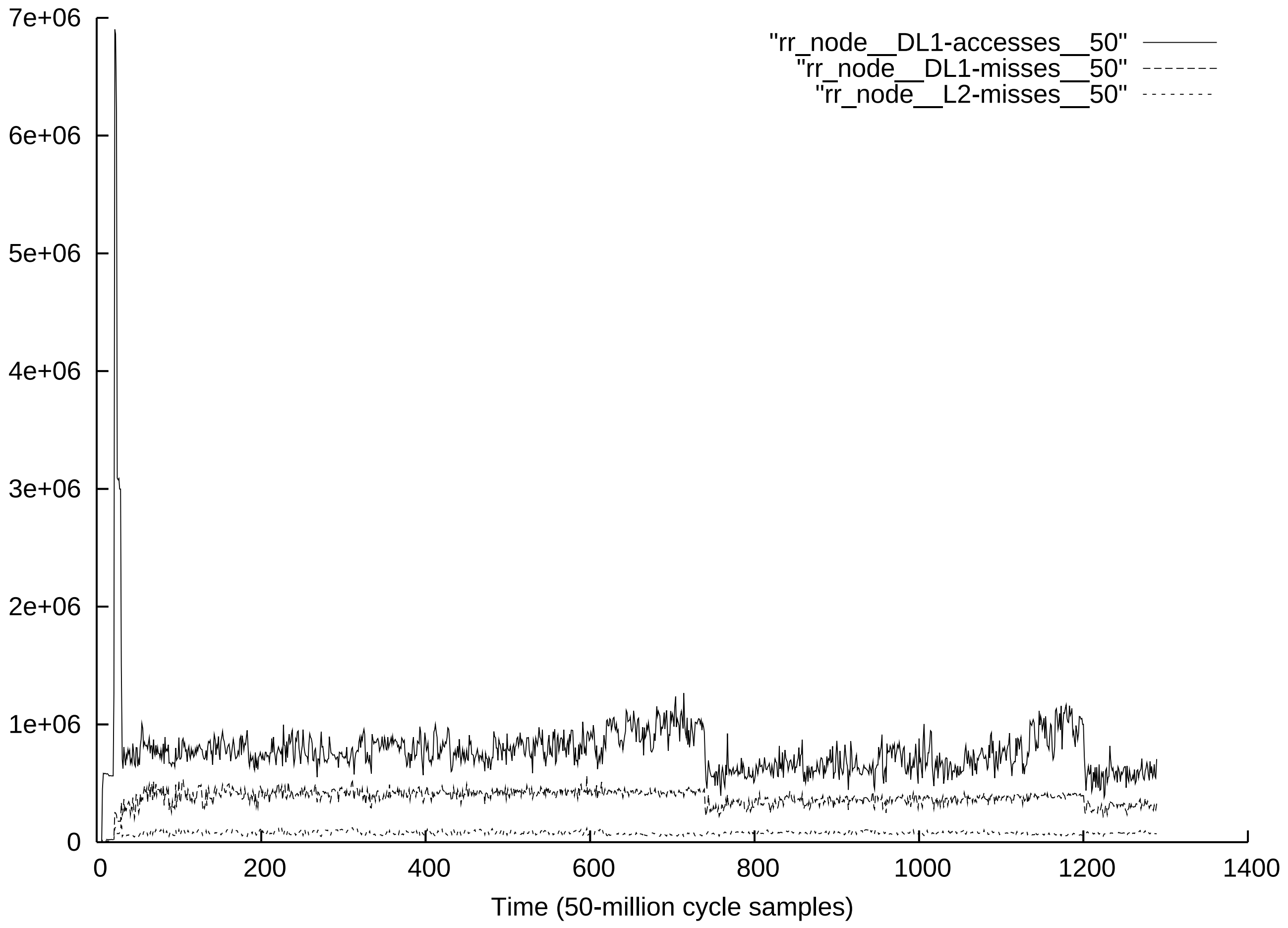


Figure 5: 175.vpr – rr_node

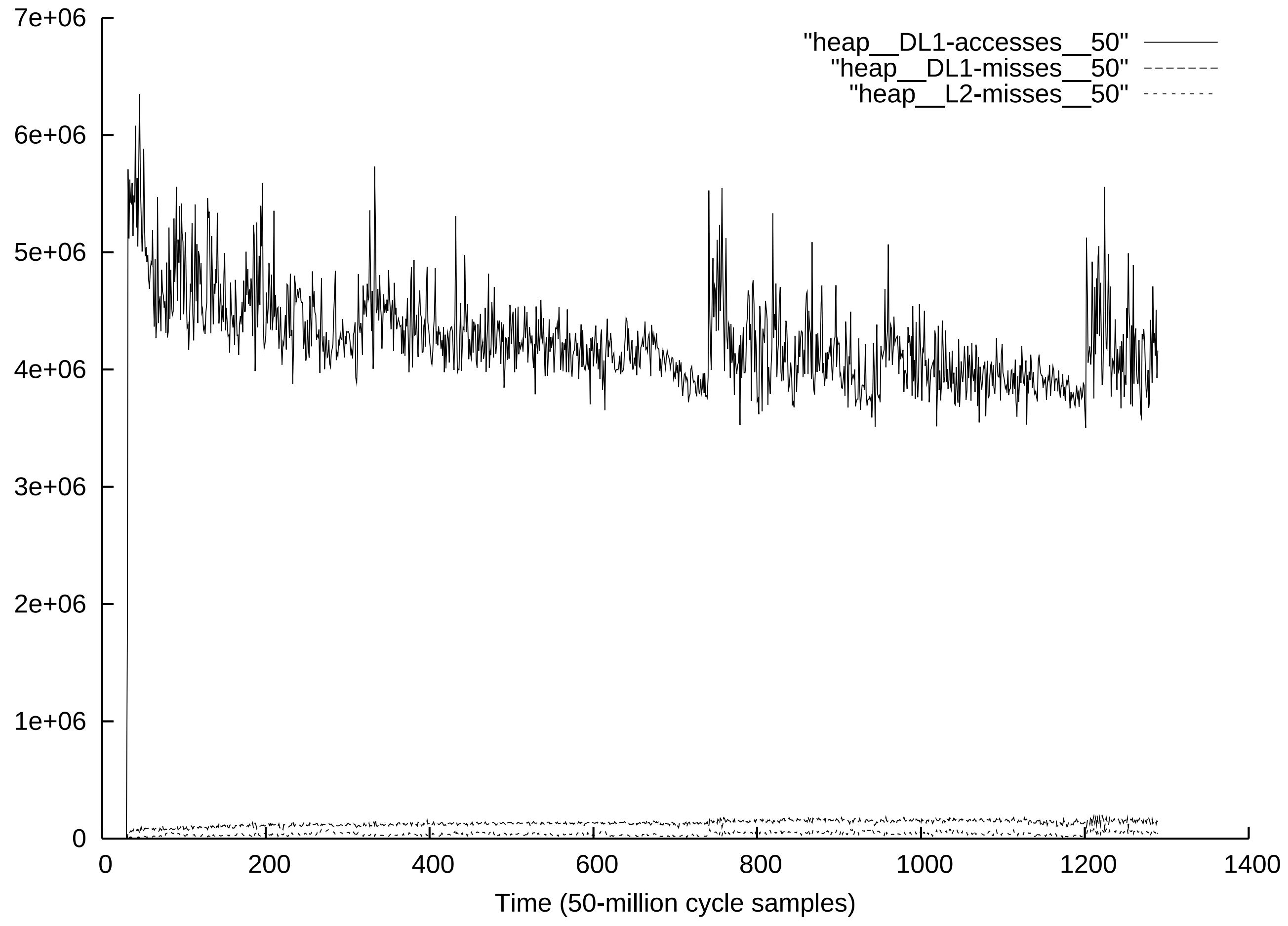


Figure 6: 175.vpr – rr_heap

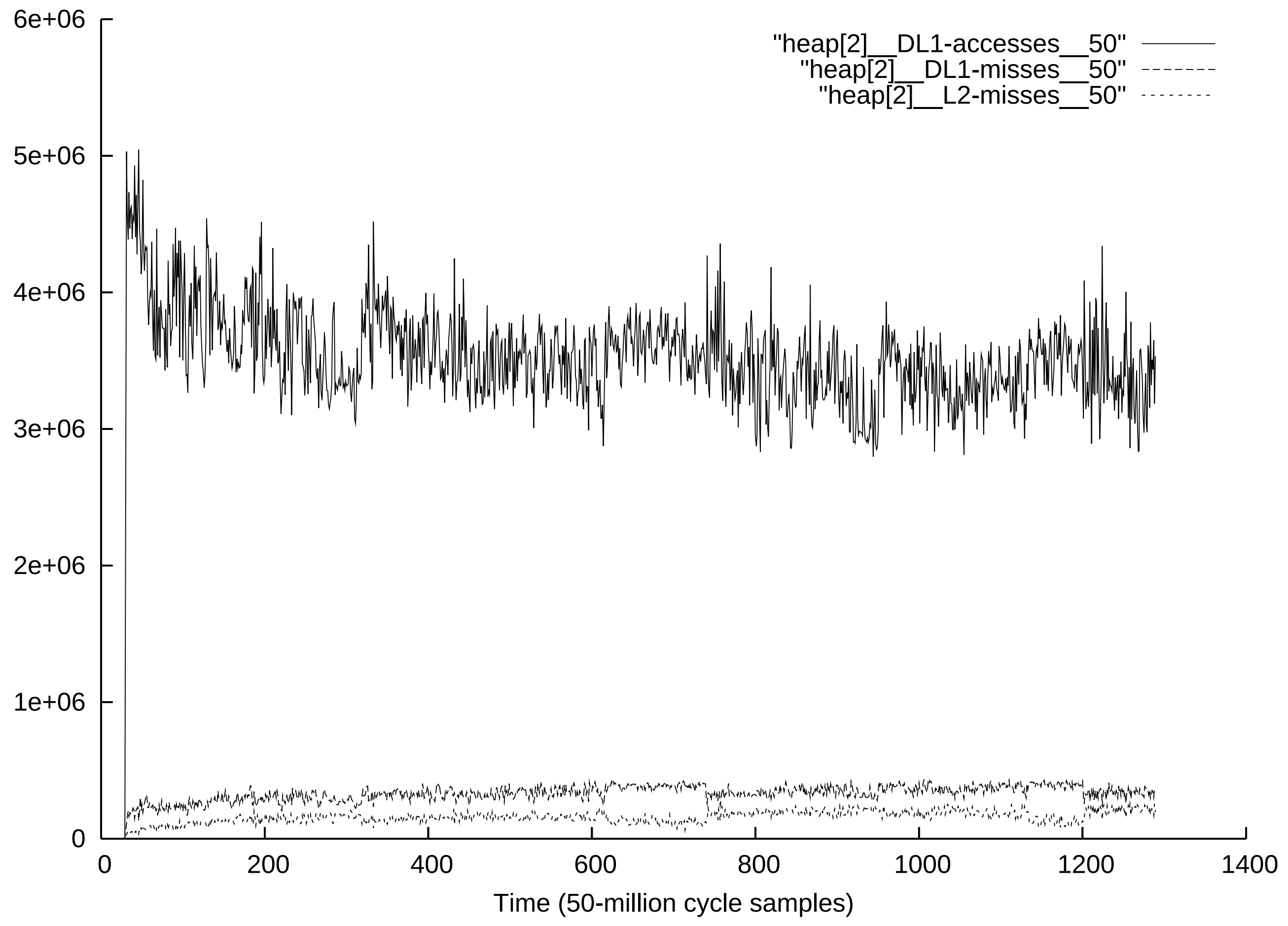


Figure 7: `175.vpr` – `rr_node_route_inf`

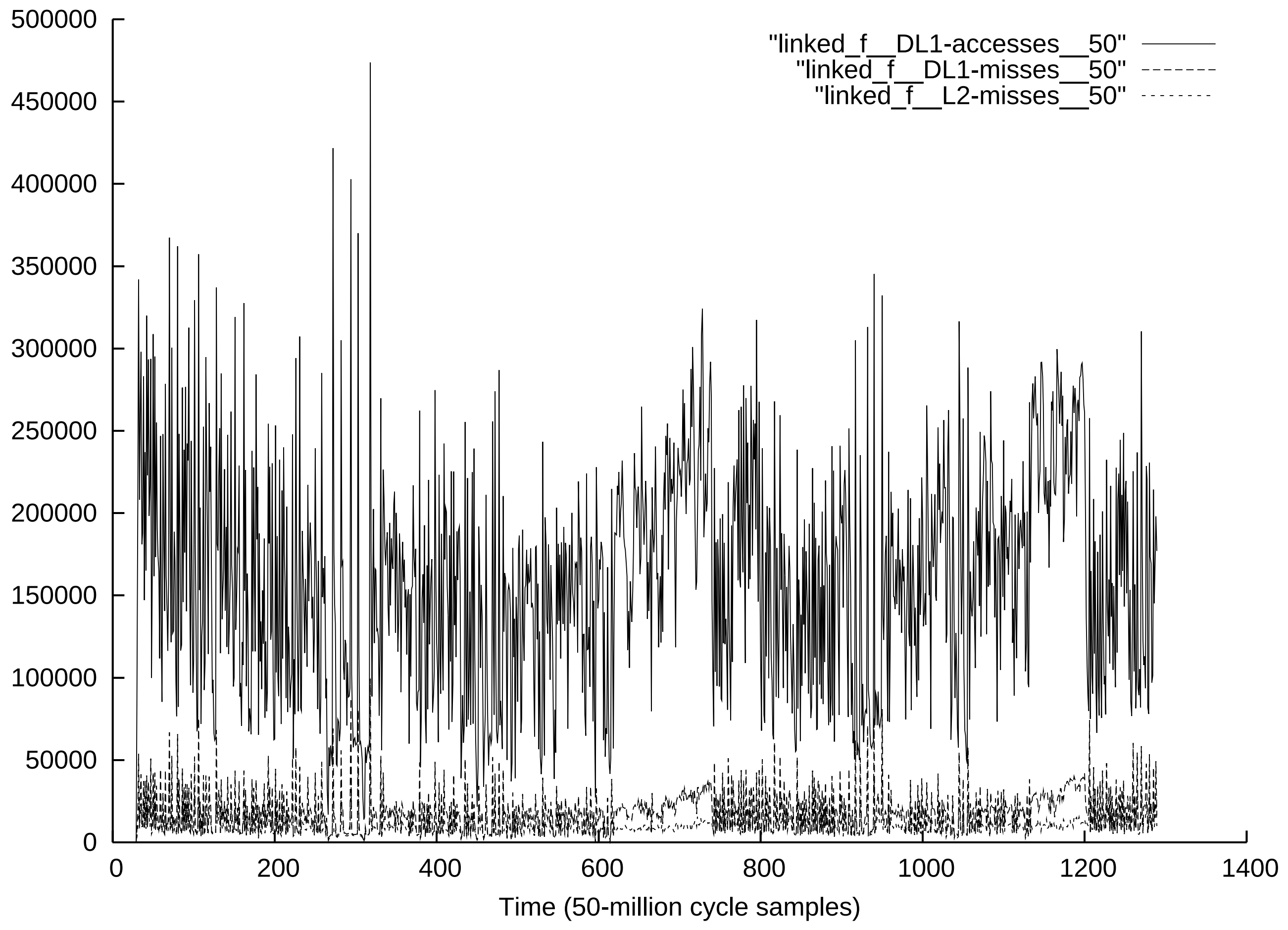


Figure 8: 175.vpr – linked_f_ptr

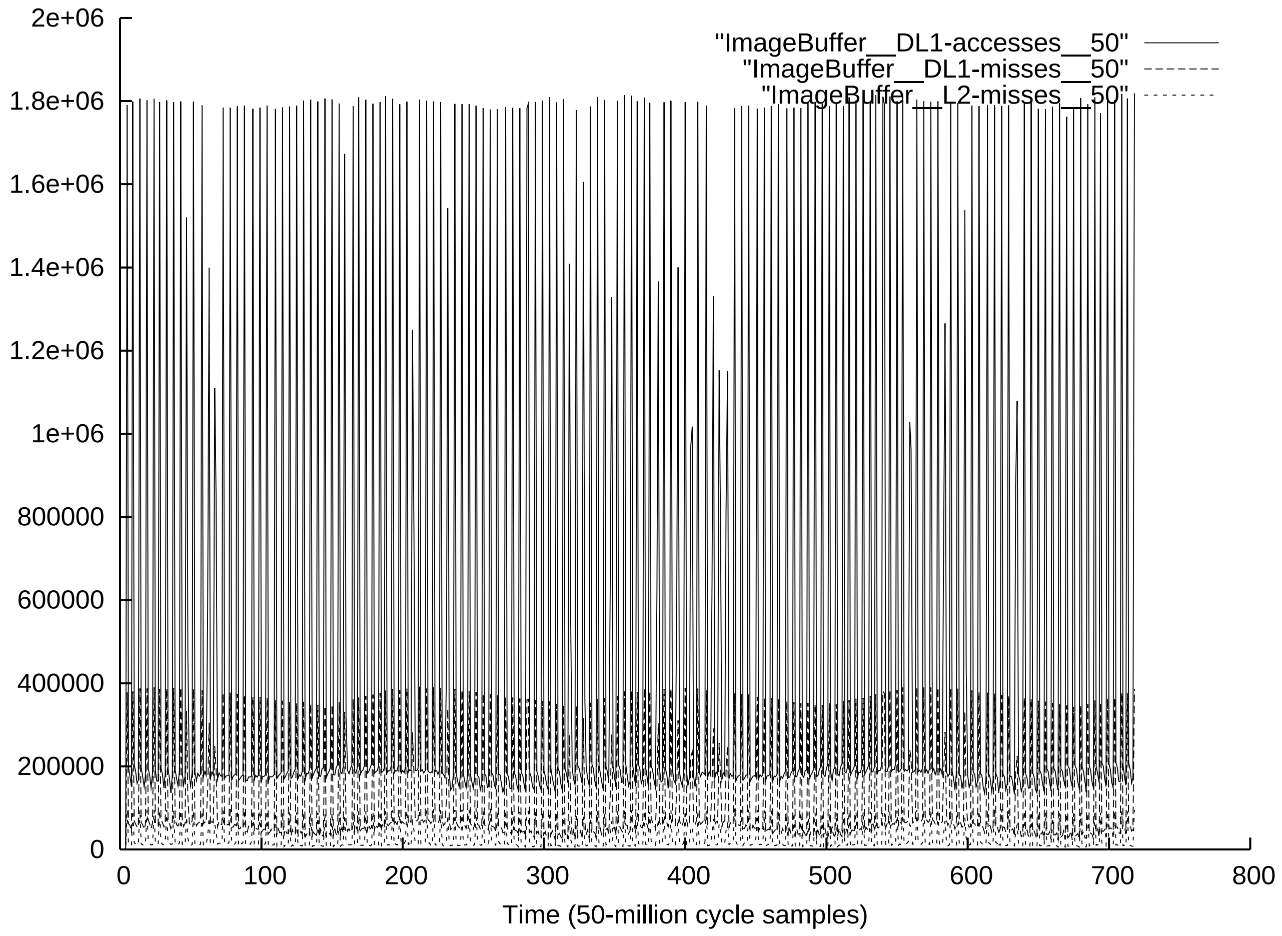


Figure 9: `177.mesa` – `Image Buffer`

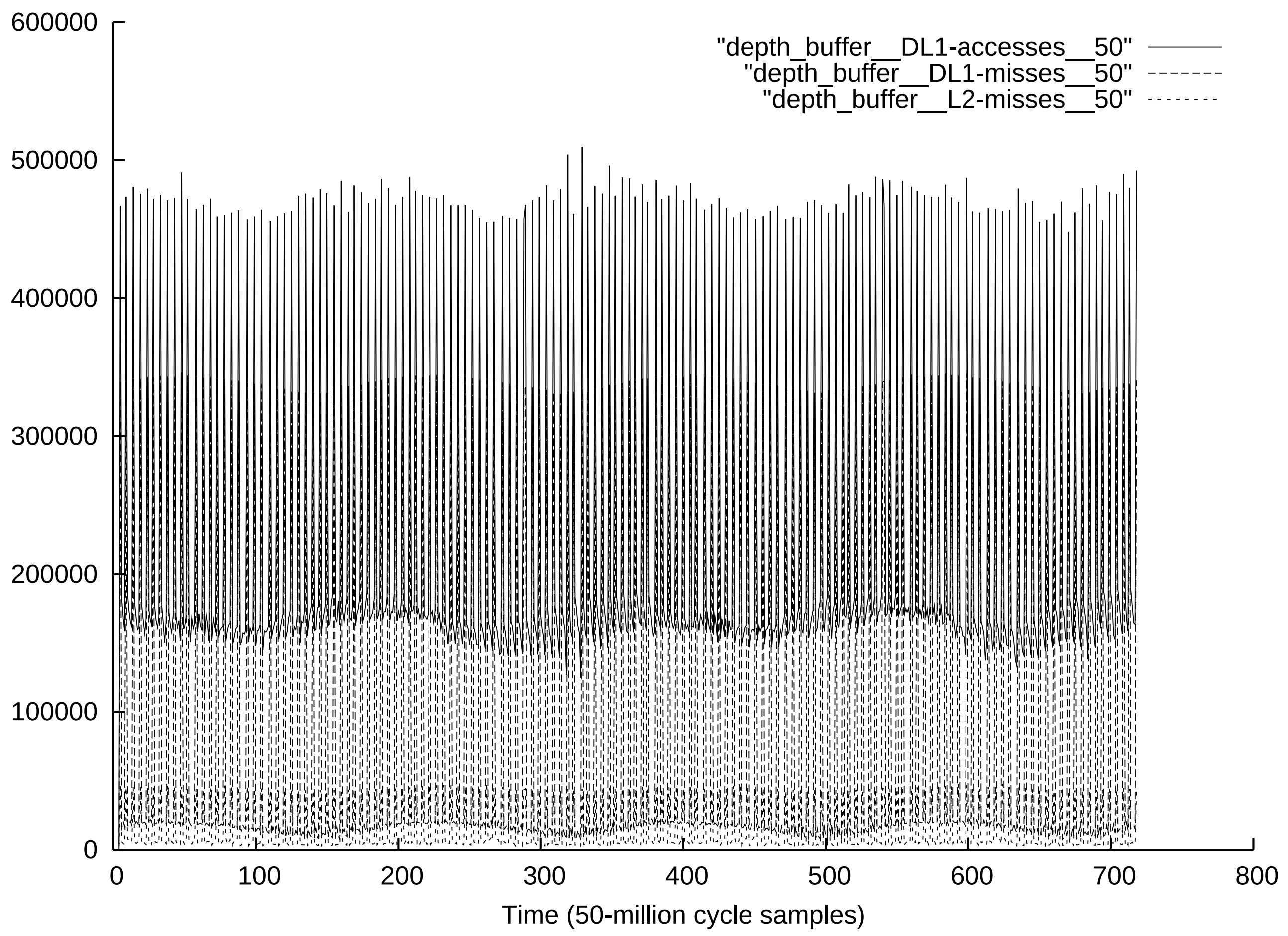


Figure 10: `177.mesa – Depth Buffer`

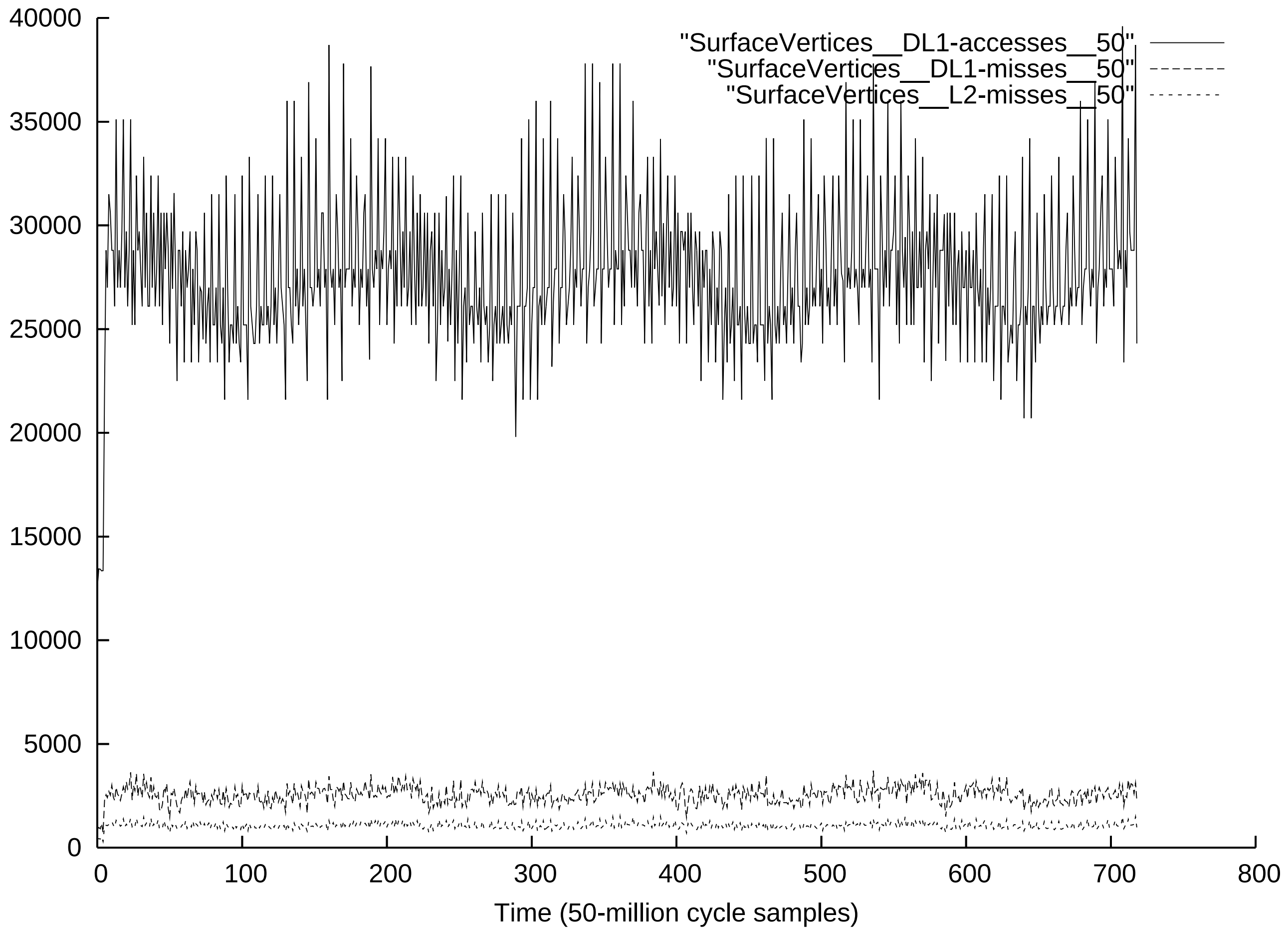


Figure 11: 177.mesa – Surface Vertex Buffer

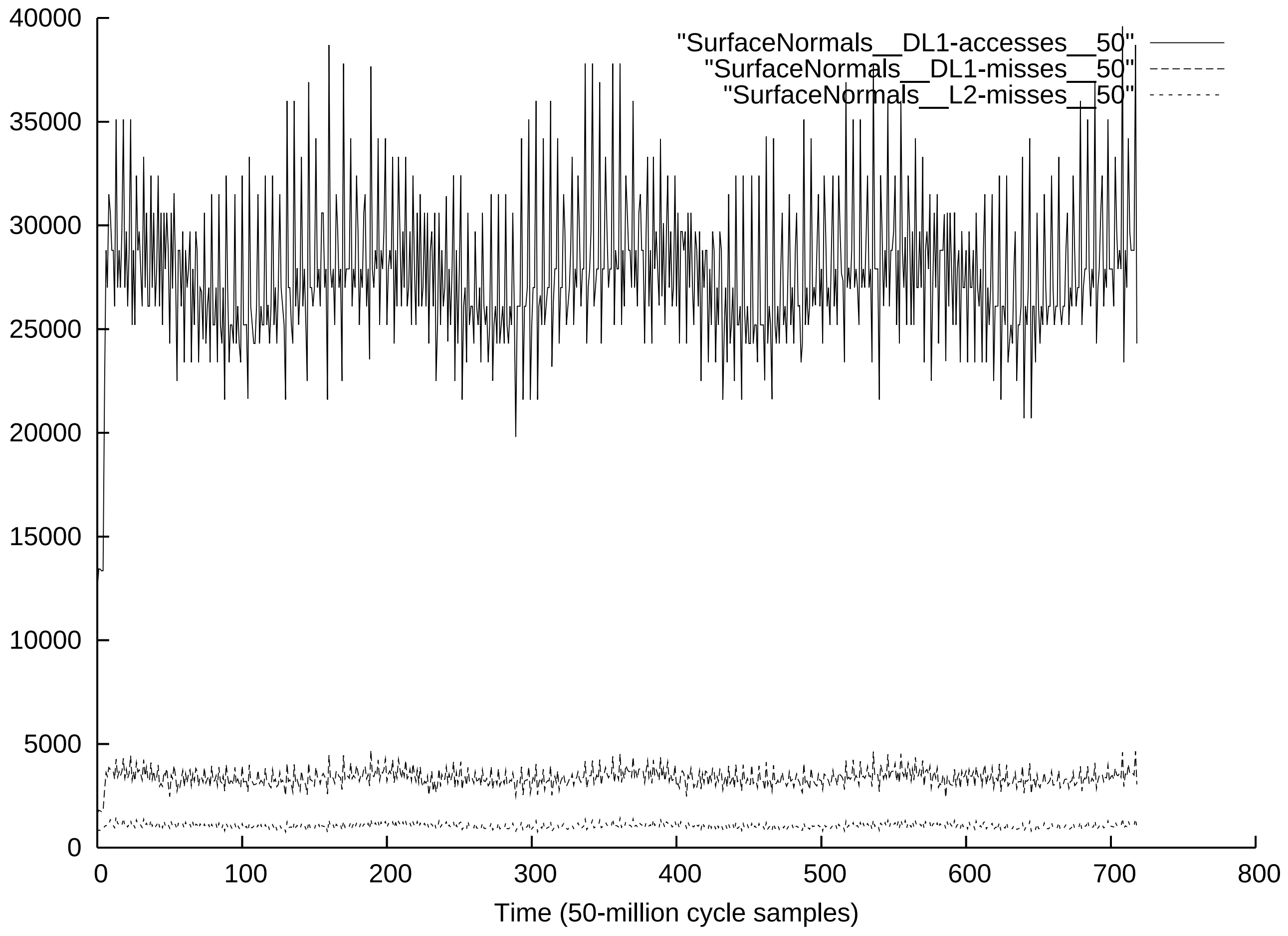


Figure 12: 177.mesa – Surface Normals

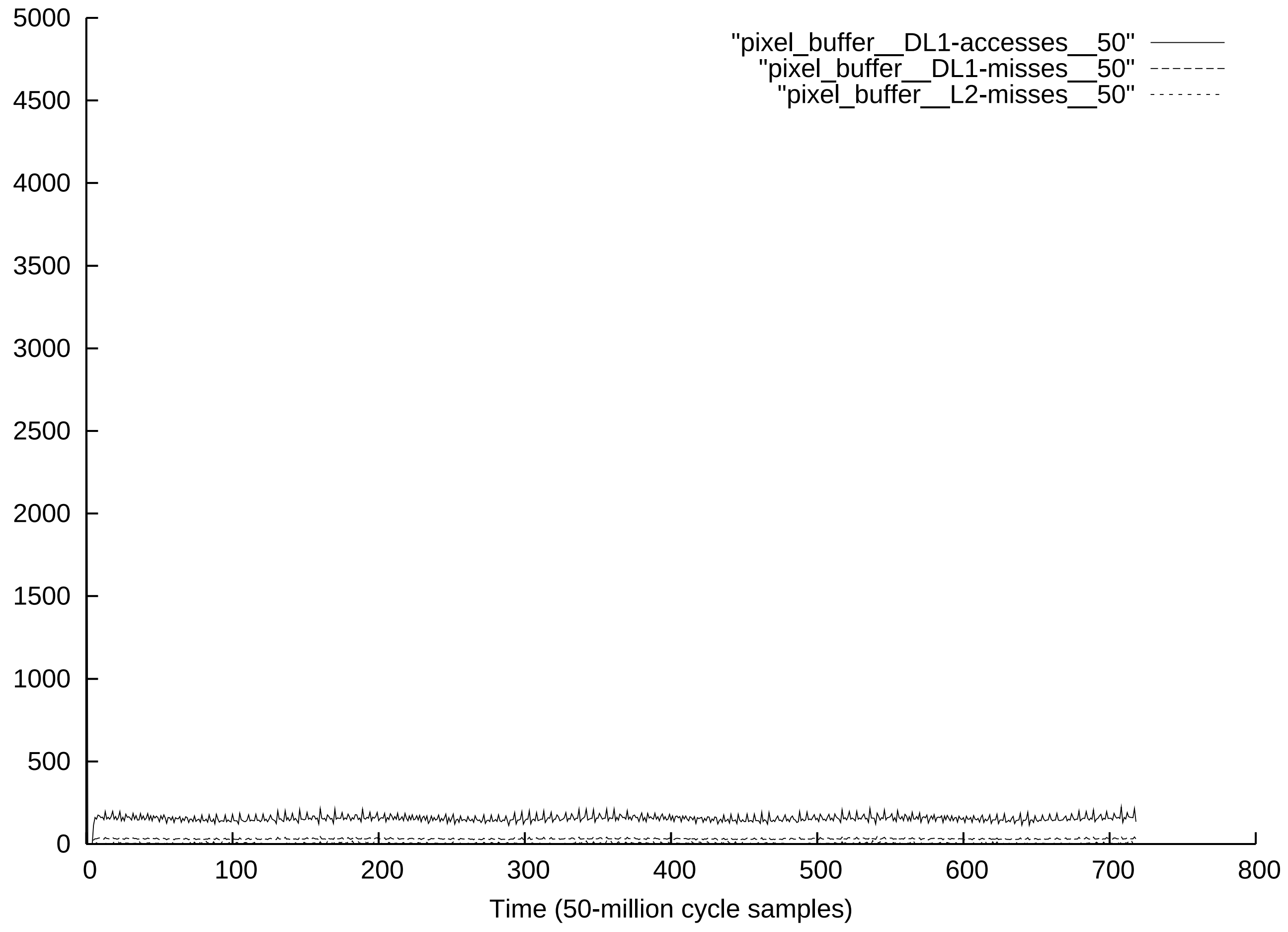


Figure 13: `177.mesa – Pixel Buffer`

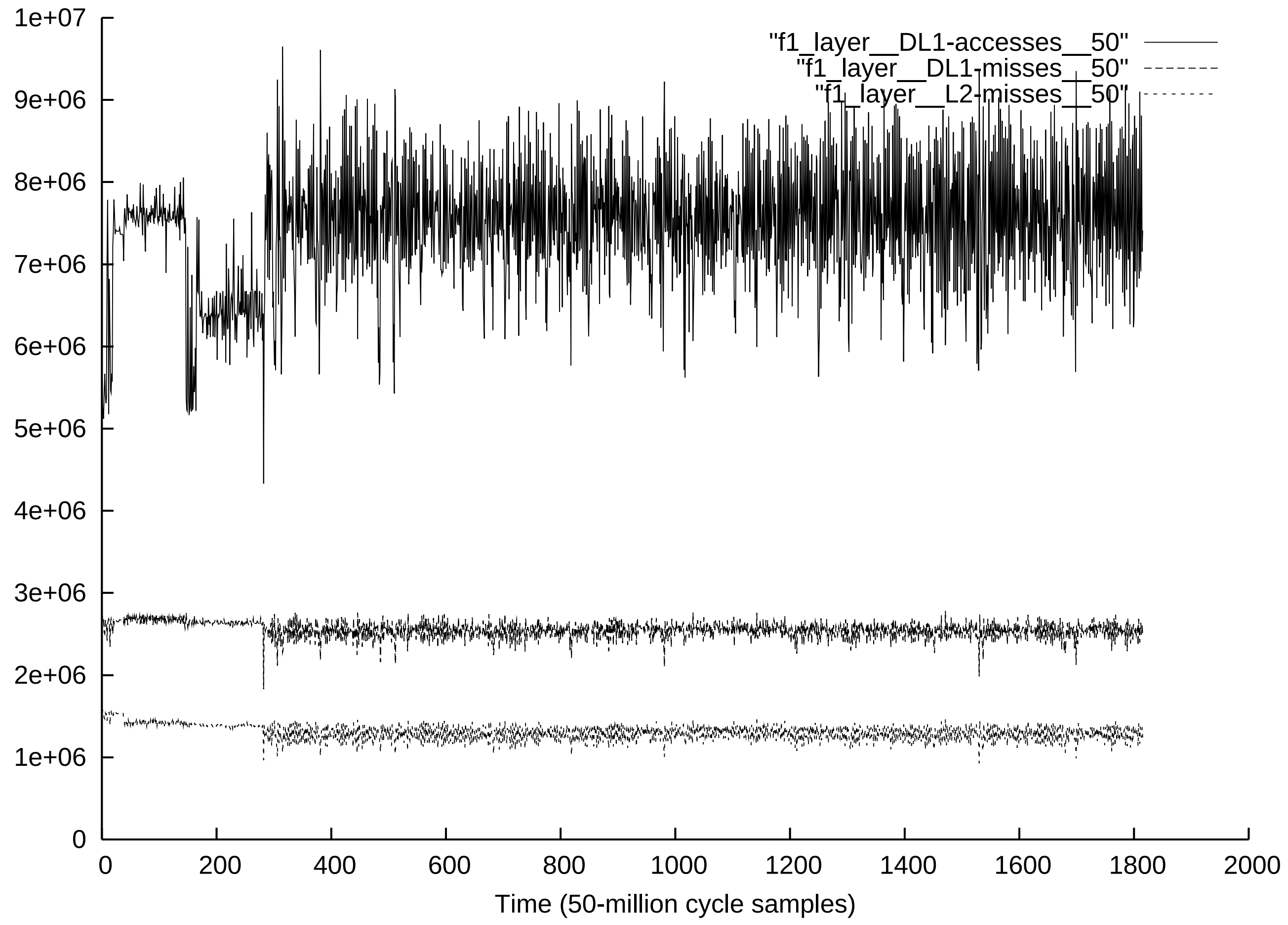


Figure 14: 179.art – f1_layer

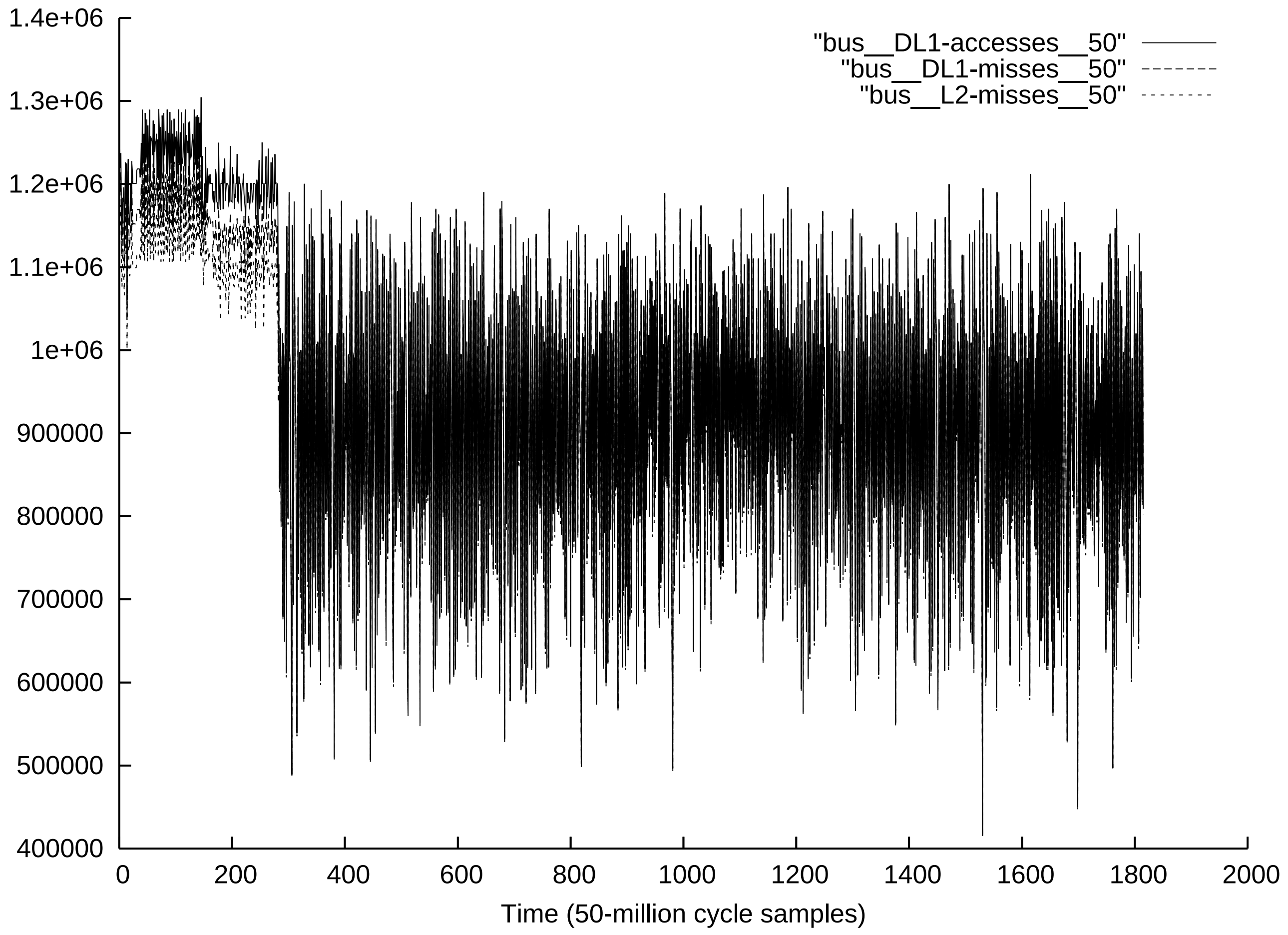


Figure 15: 179.art – bus

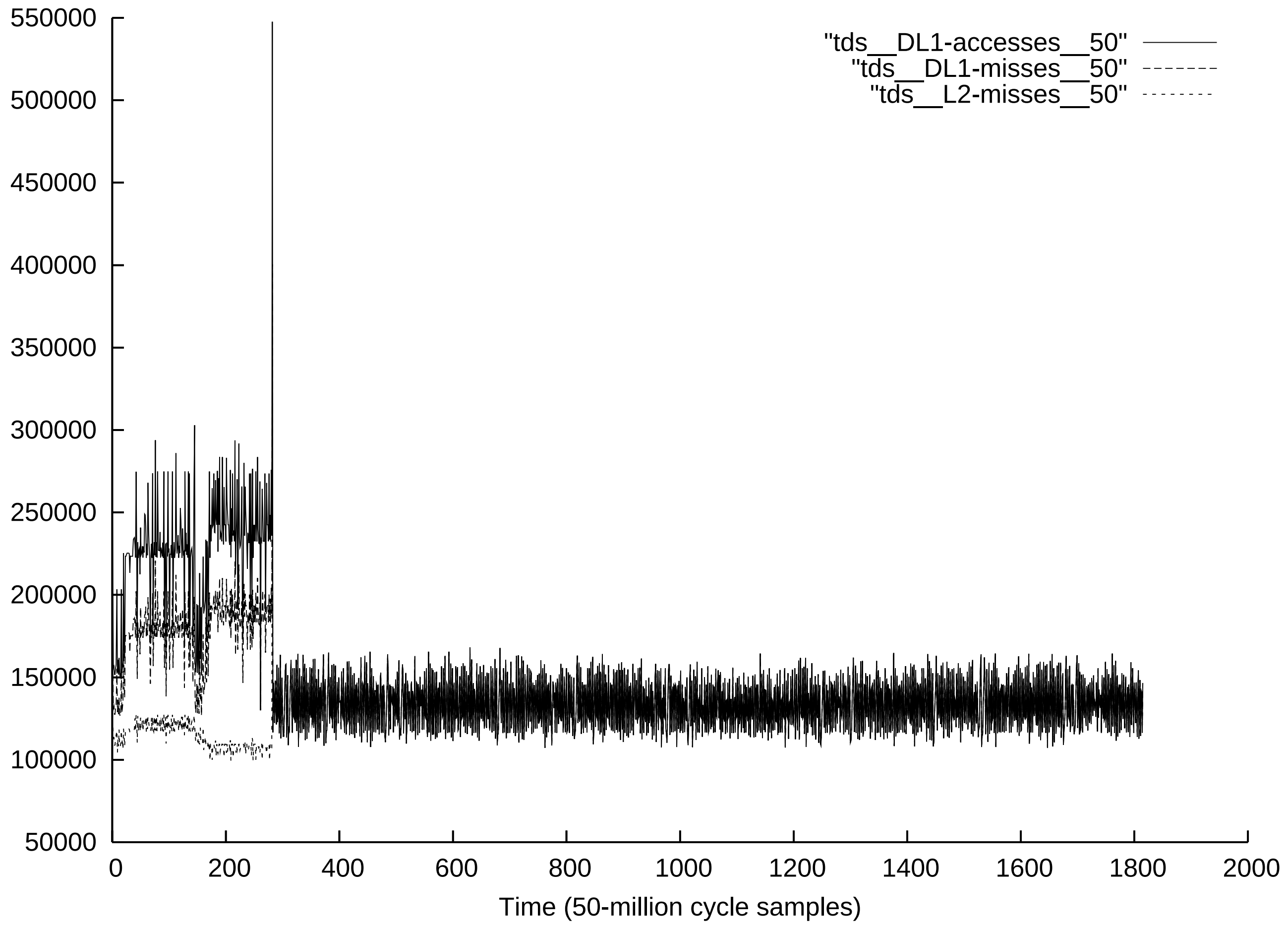


Figure 16: 179.art – tds

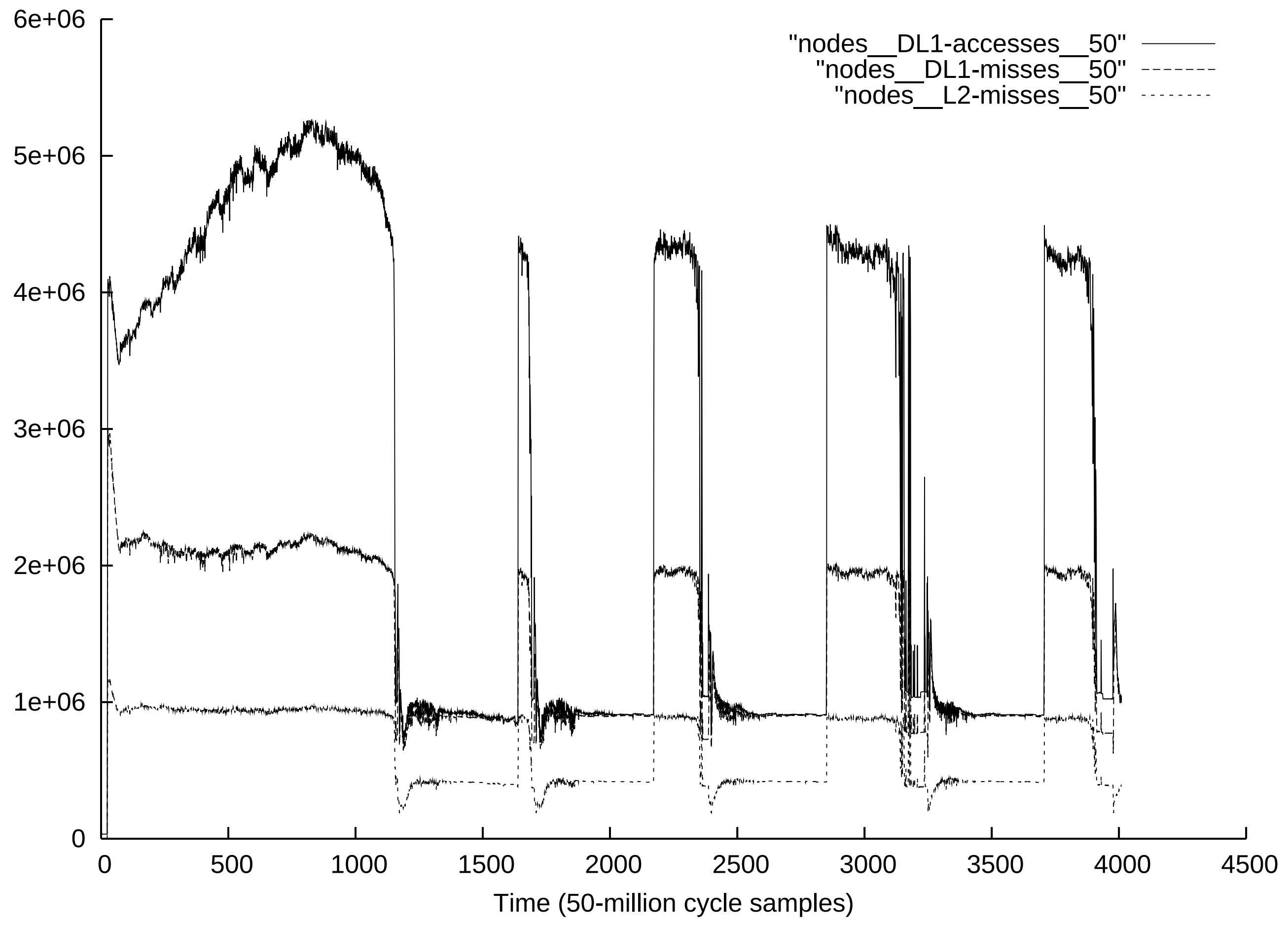


Figure 17: 181.mcf – nodes

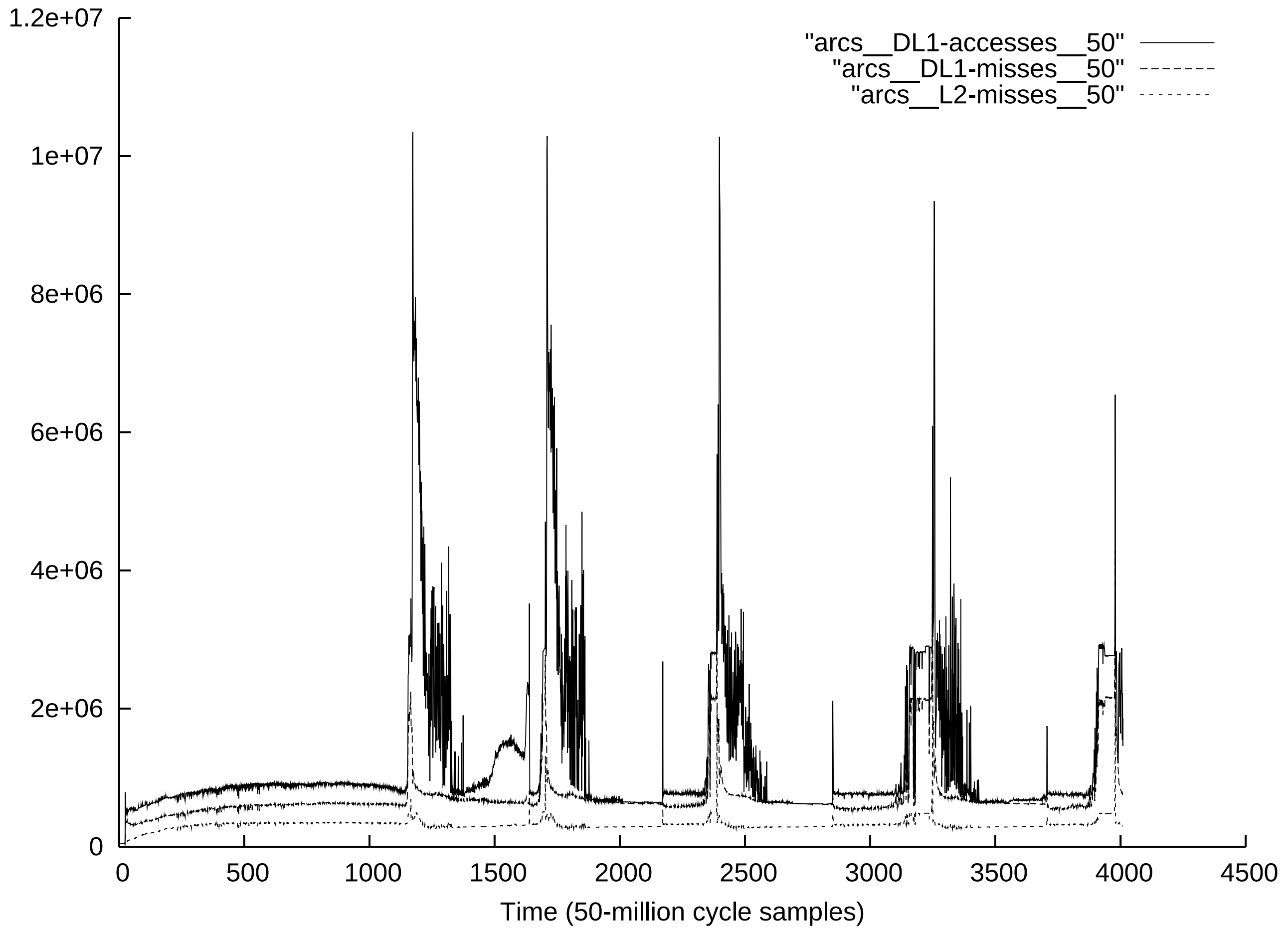


Figure 18: 181.mcf – arcs

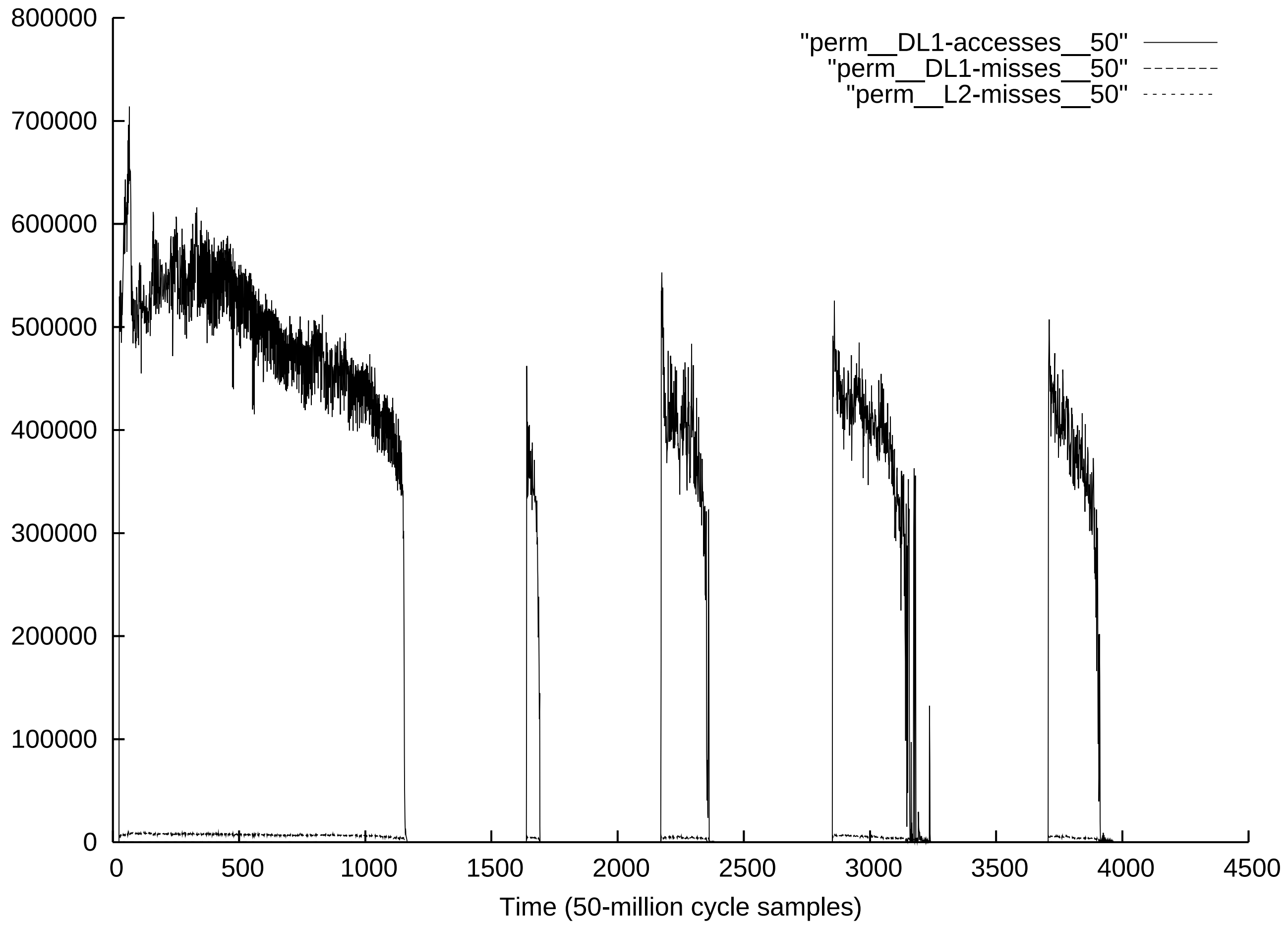
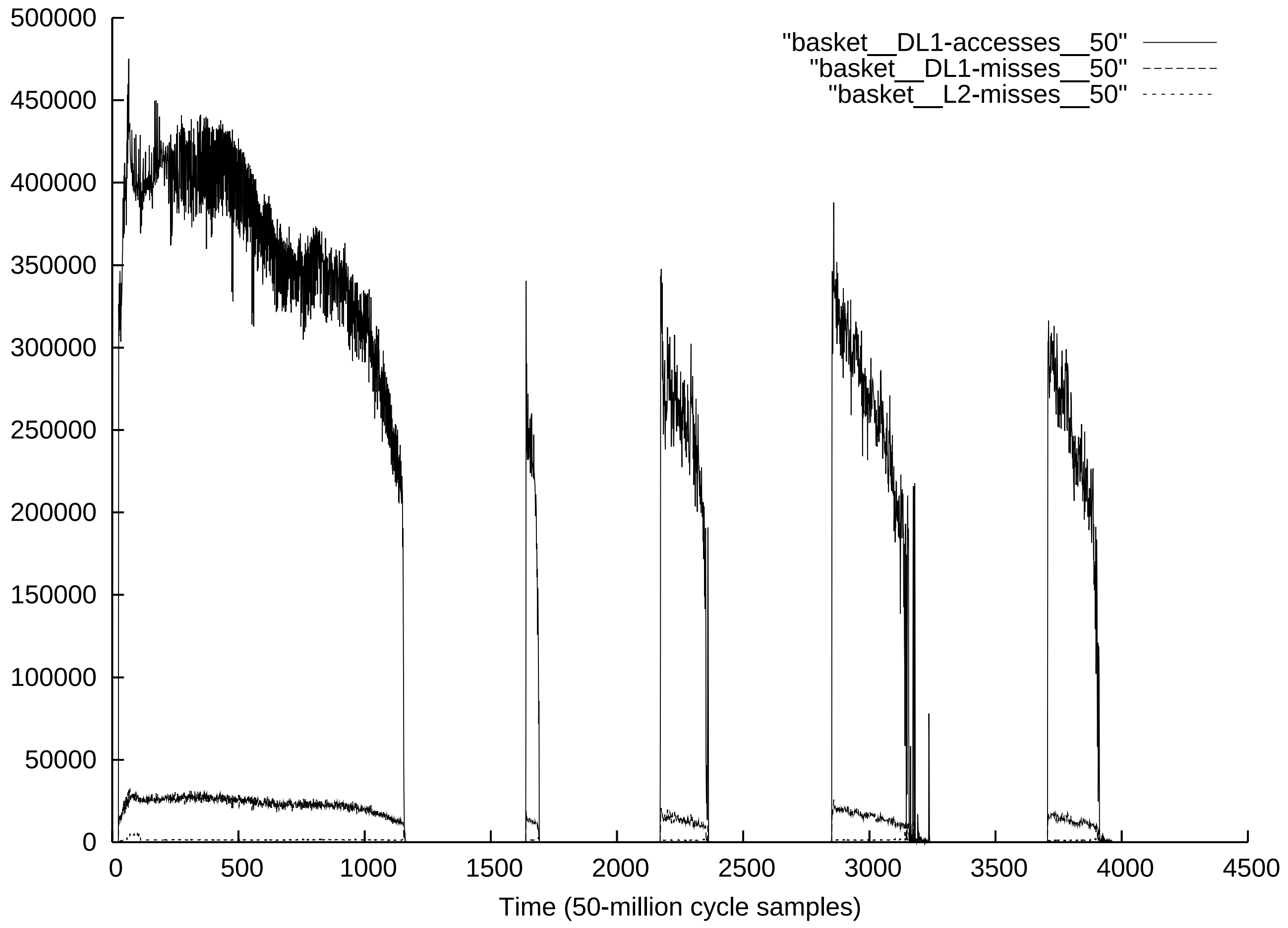


Figure 19: 181.mcf – perm

Figure 20: 181.mcf – basket



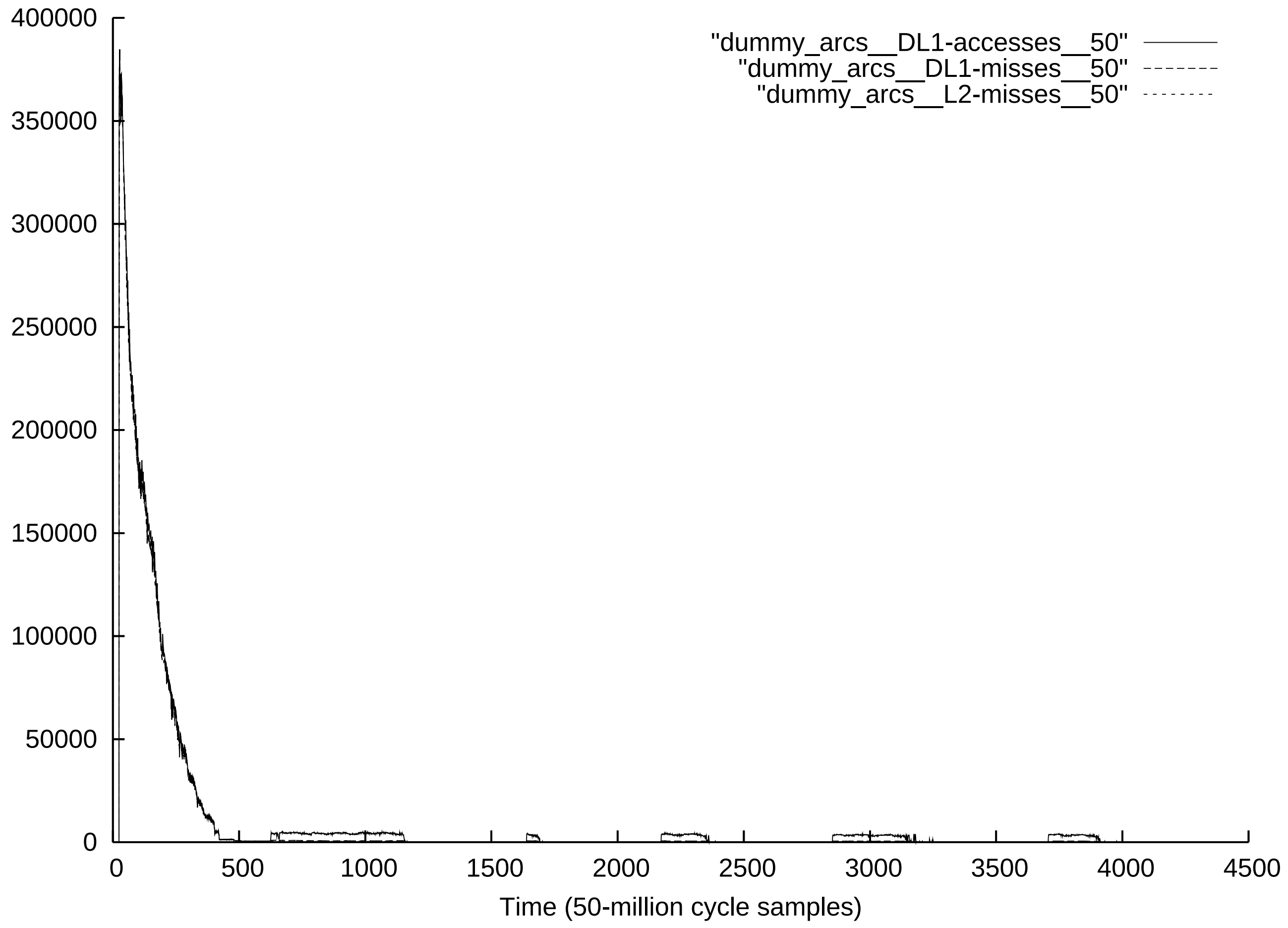


Figure 21: 181.mcf – dummy_arcs

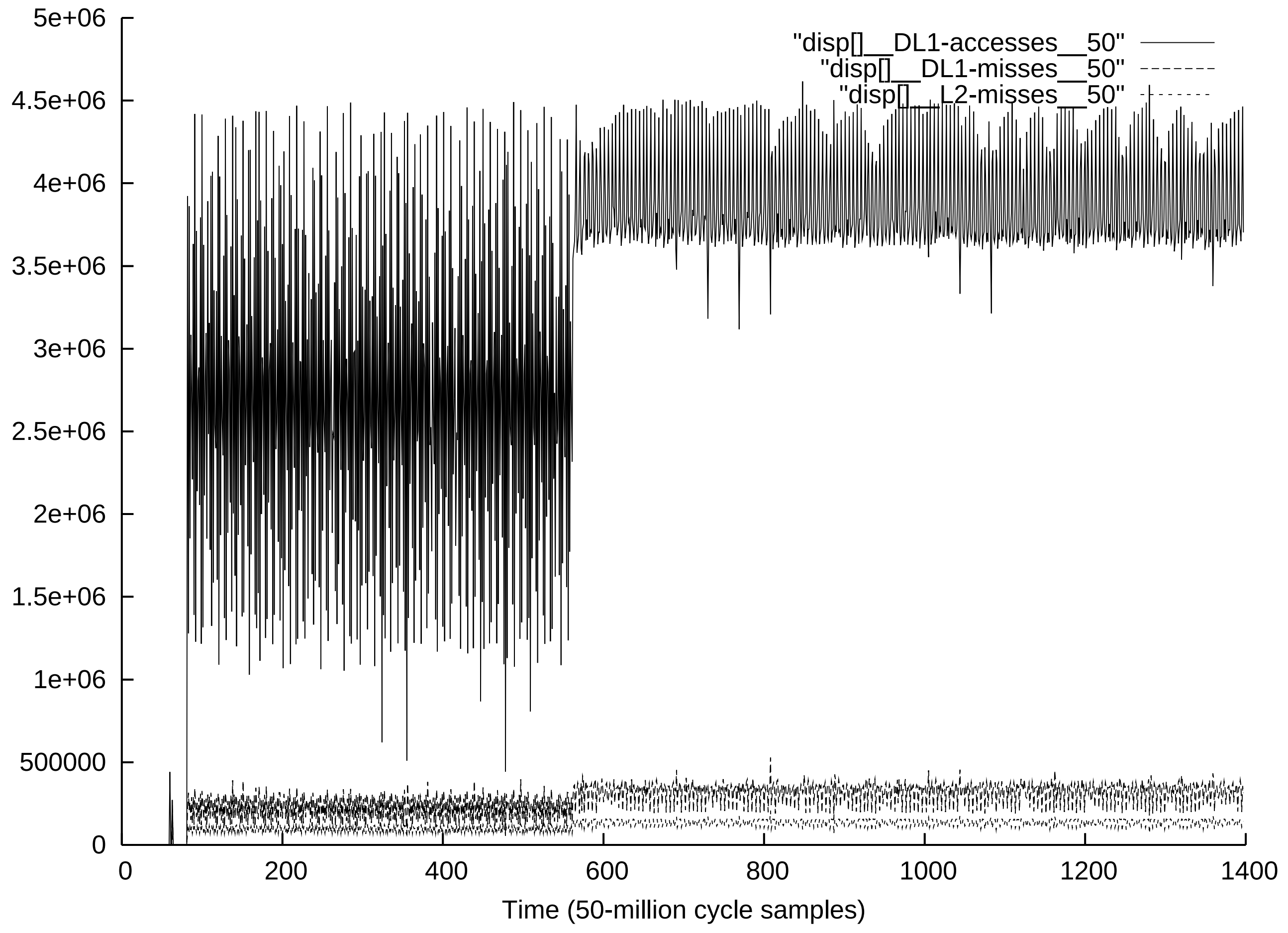


Figure 22: 183.equake – K[] []

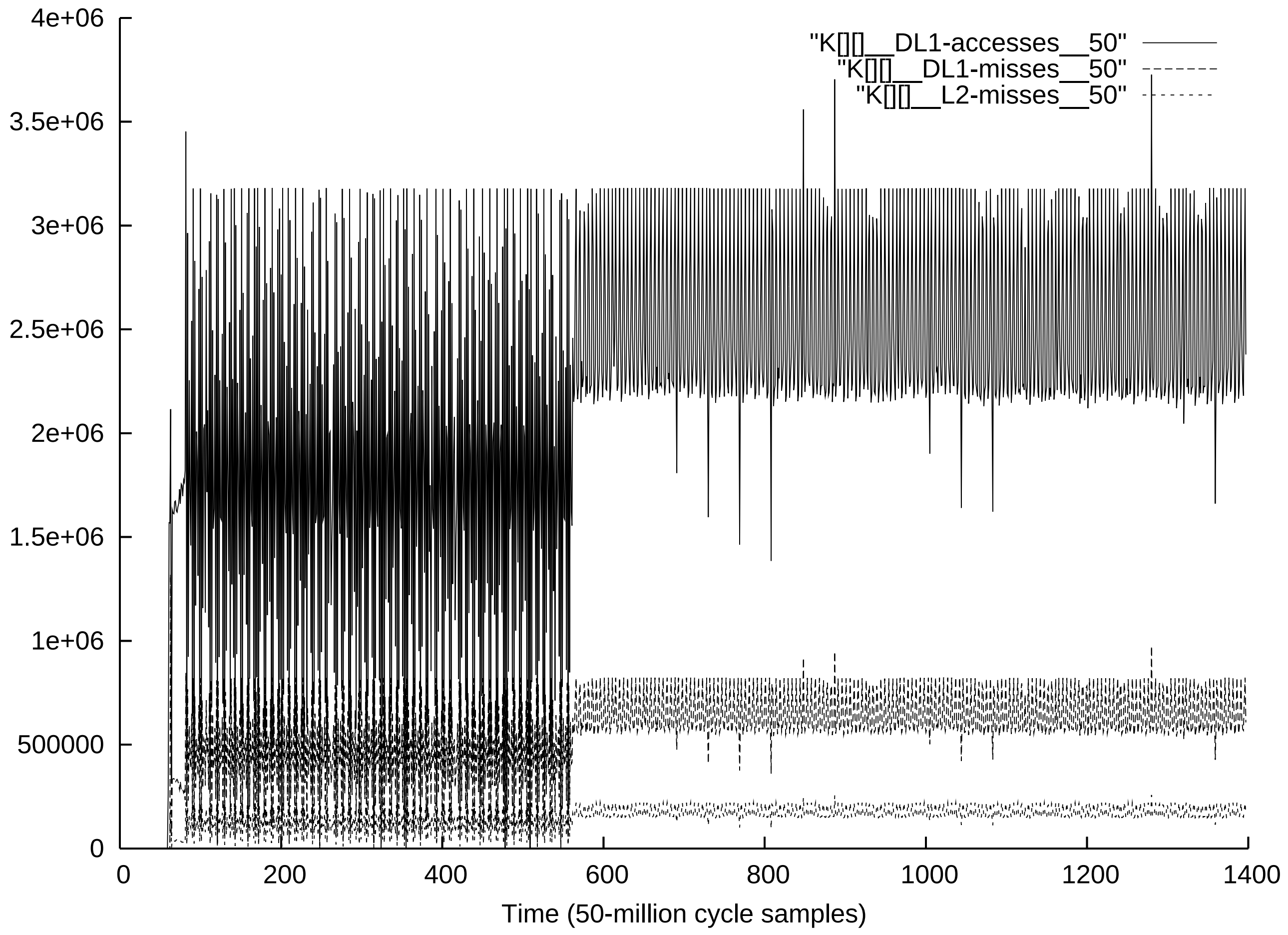


Figure 23: 183.equake – disp[]

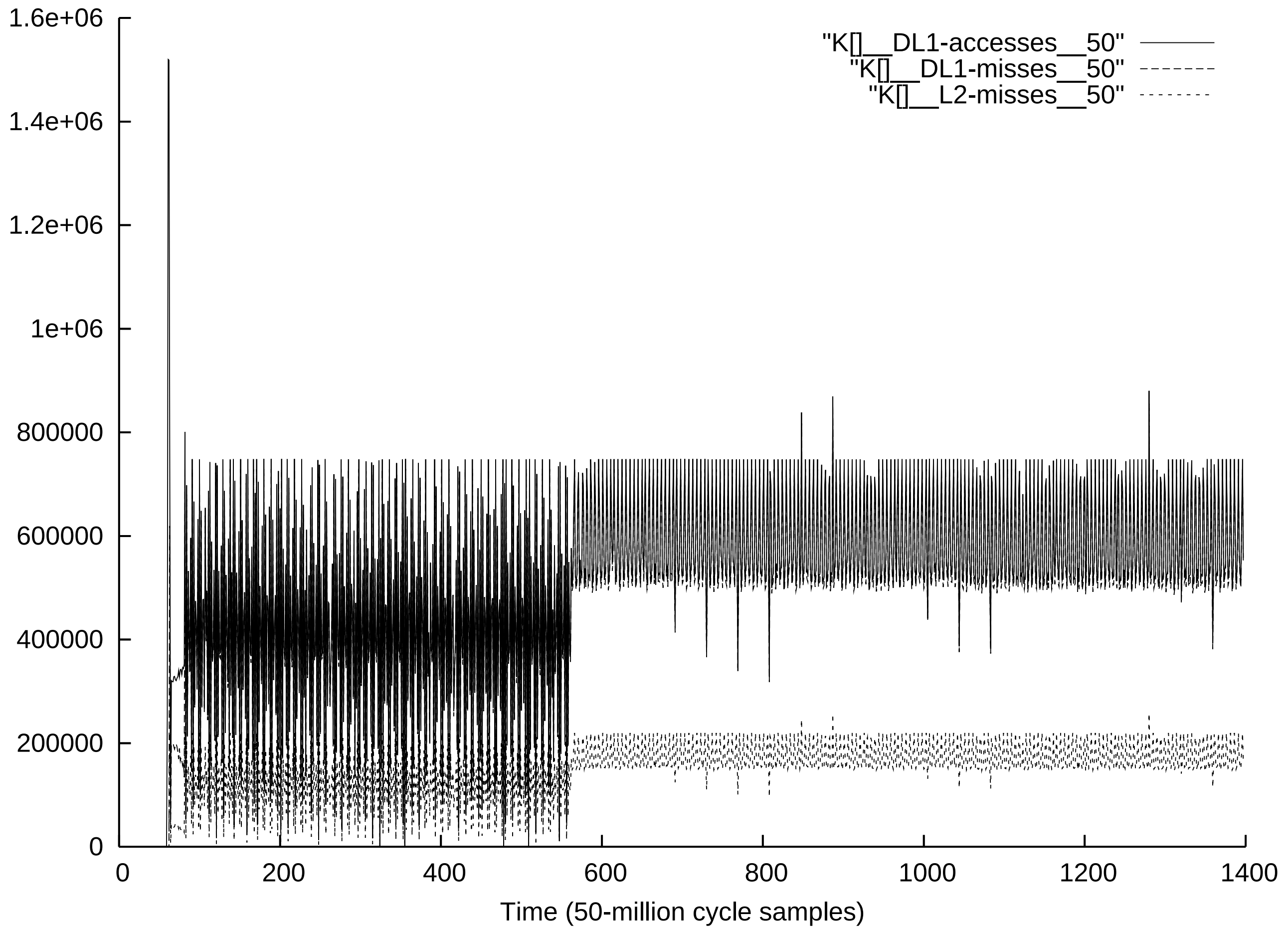


Figure 24: 183.equake – K[]

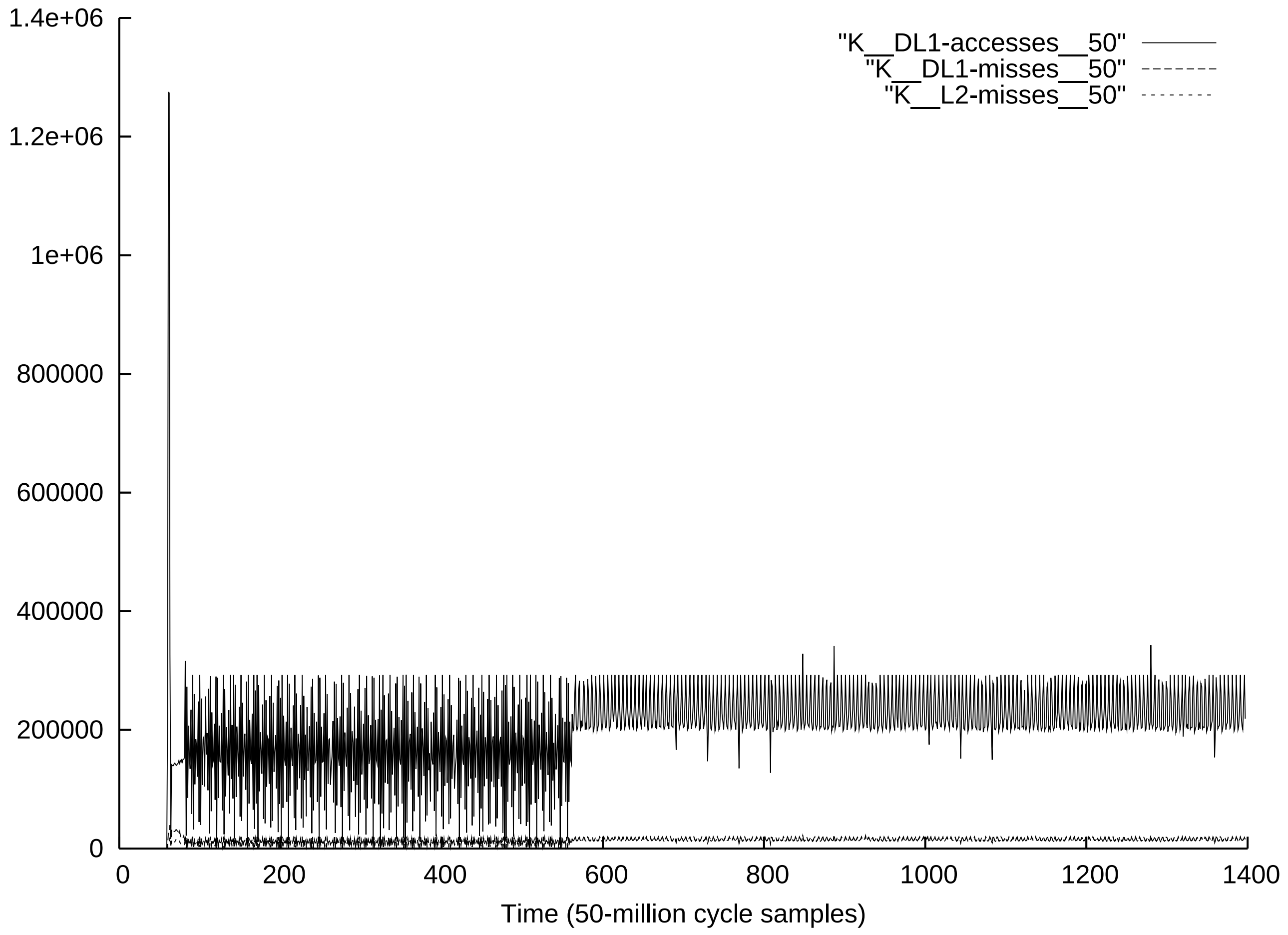


Figure 25: 183.equake – K

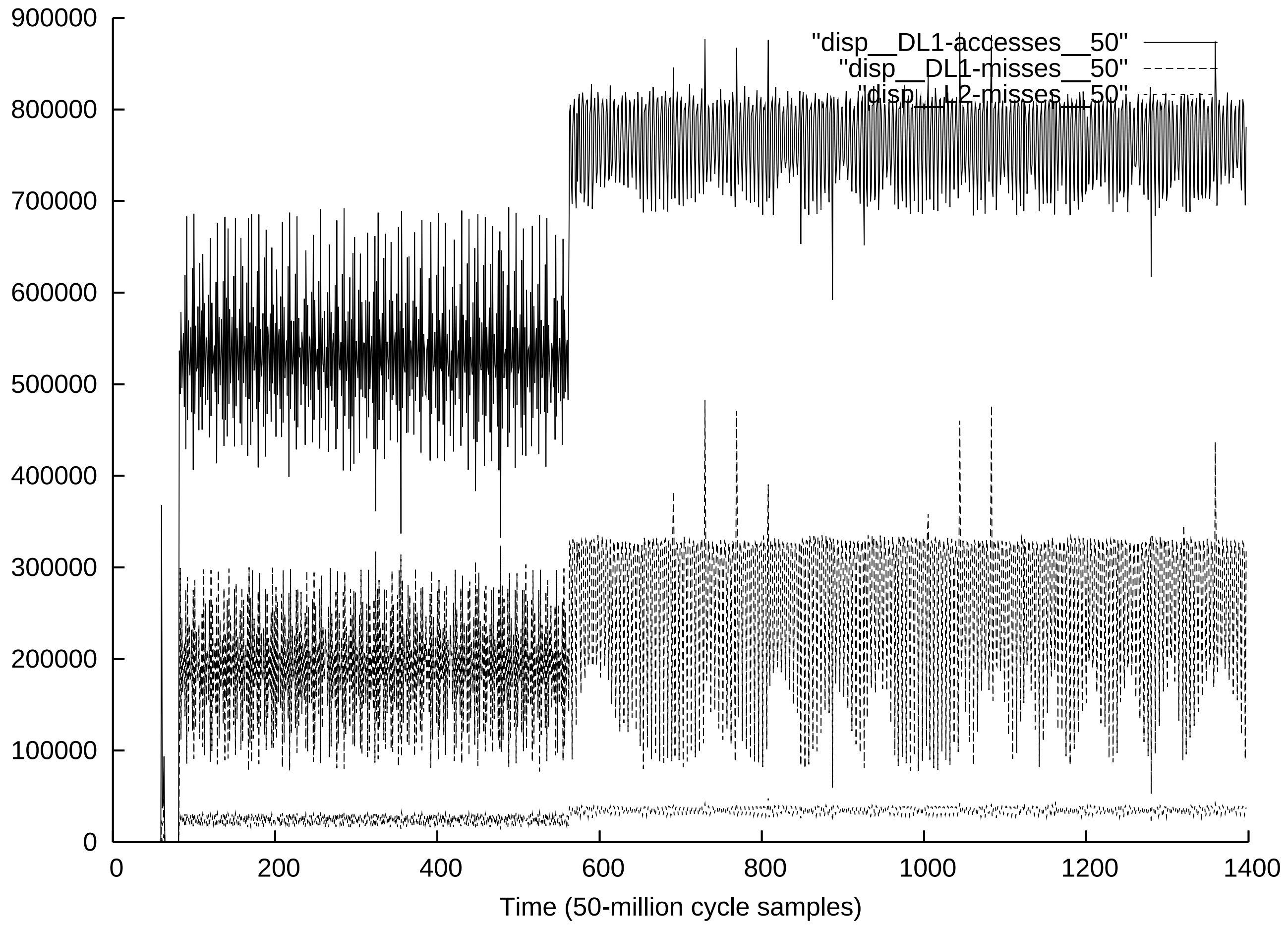


Figure 26: 183.equake – disp

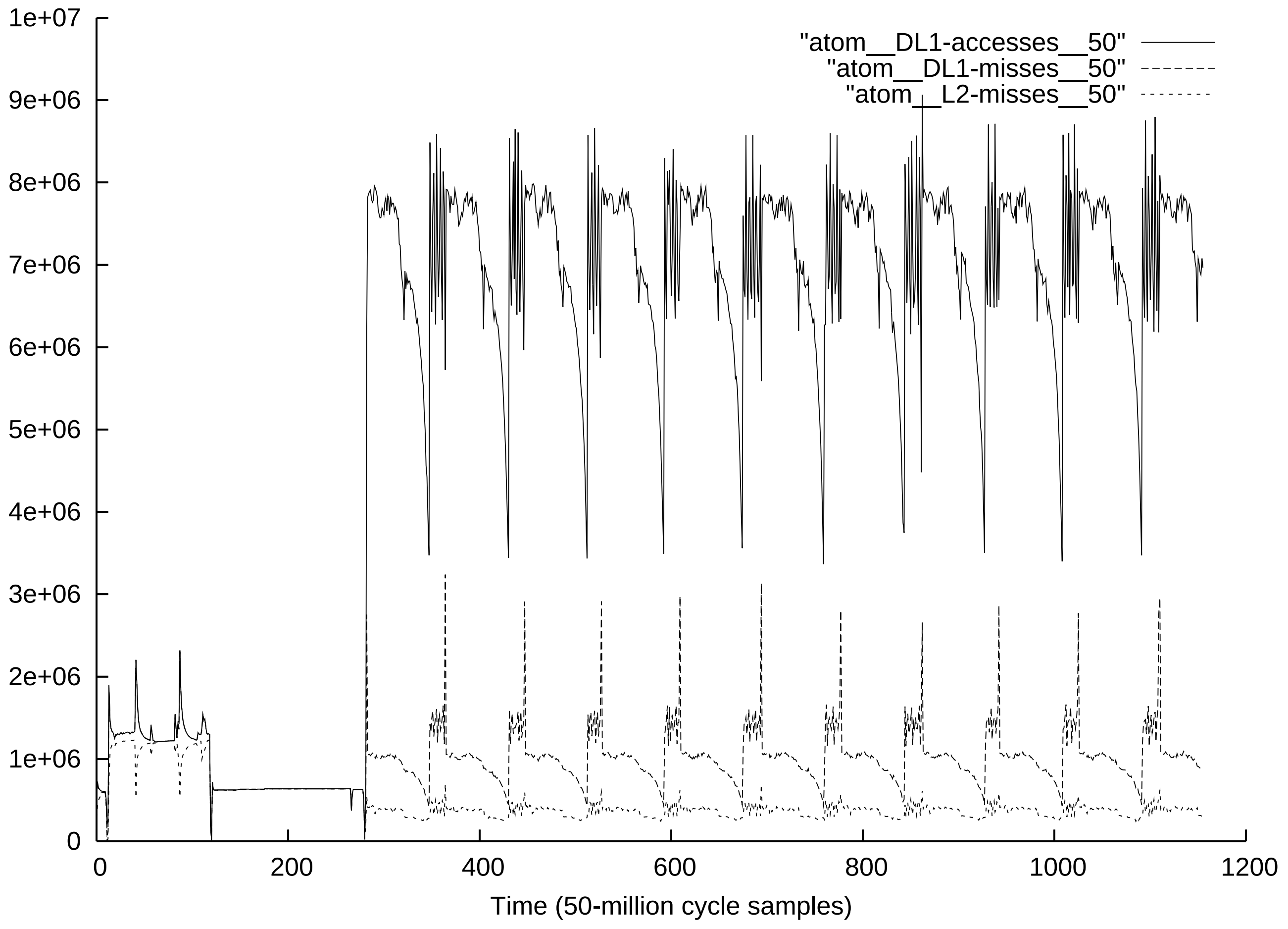
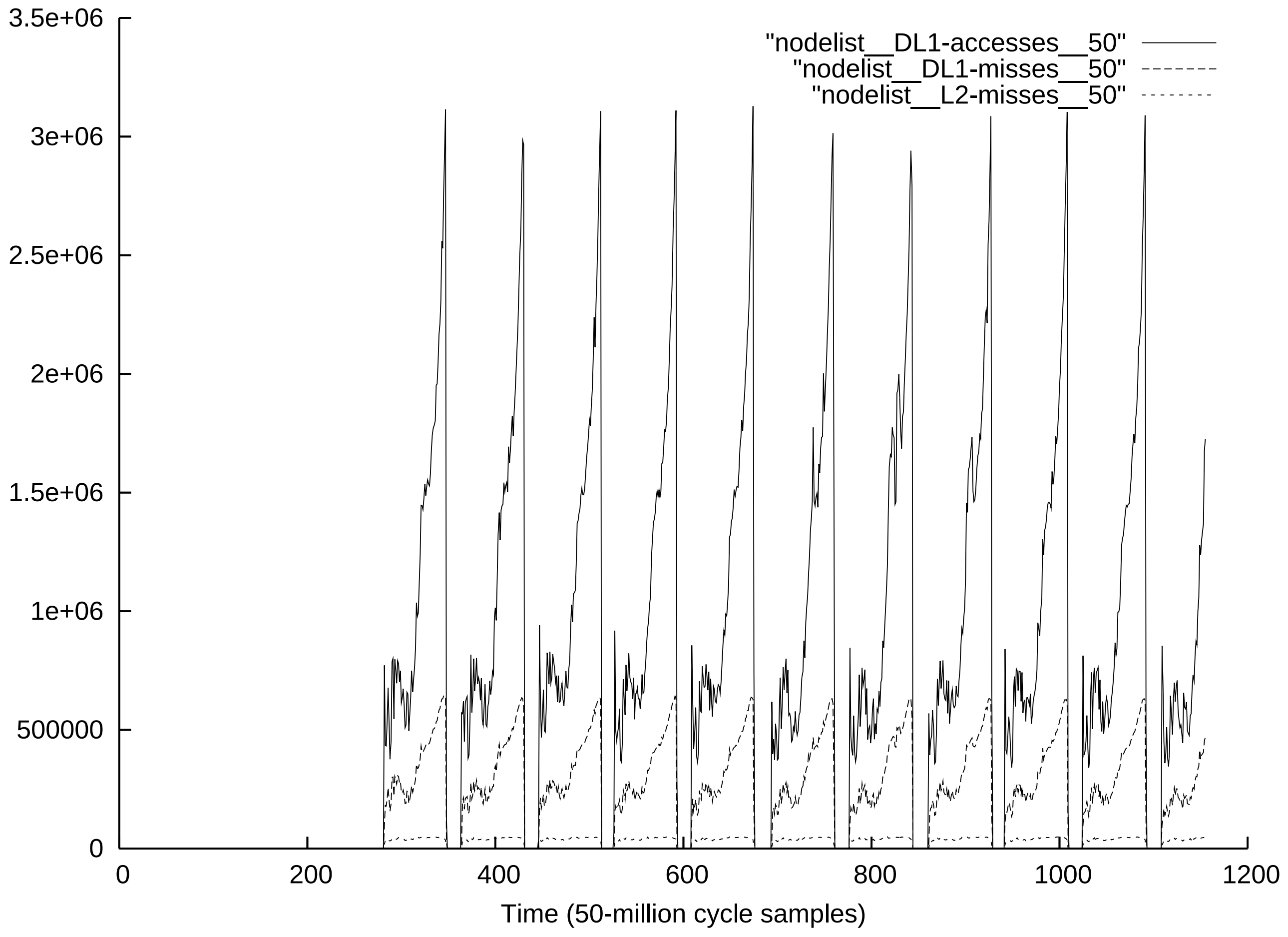


Figure 27: 188.ammp – atom

Figure 28: `188.ammp – nodelist`



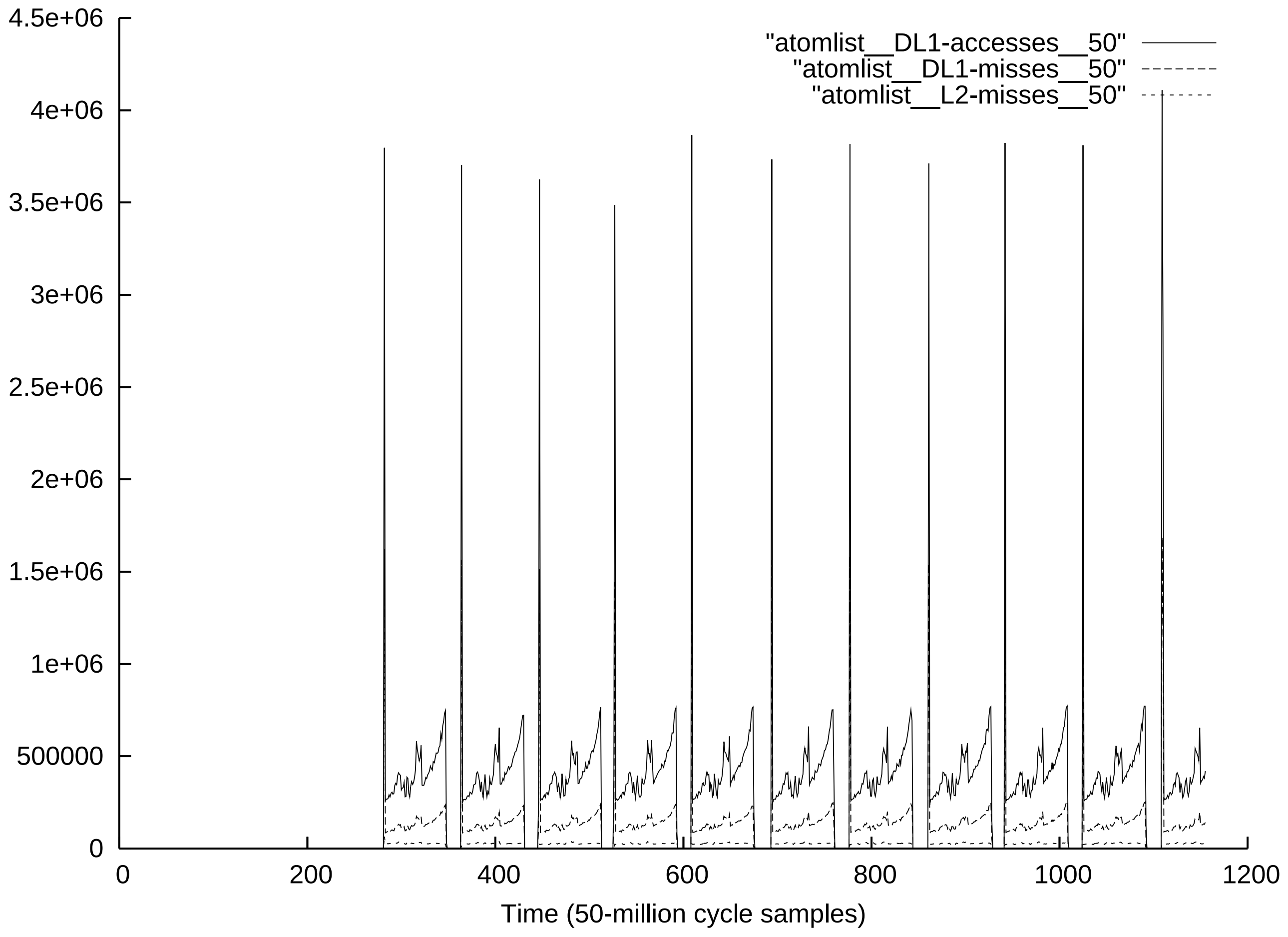


Figure 29: `188.ammp – atomlist`

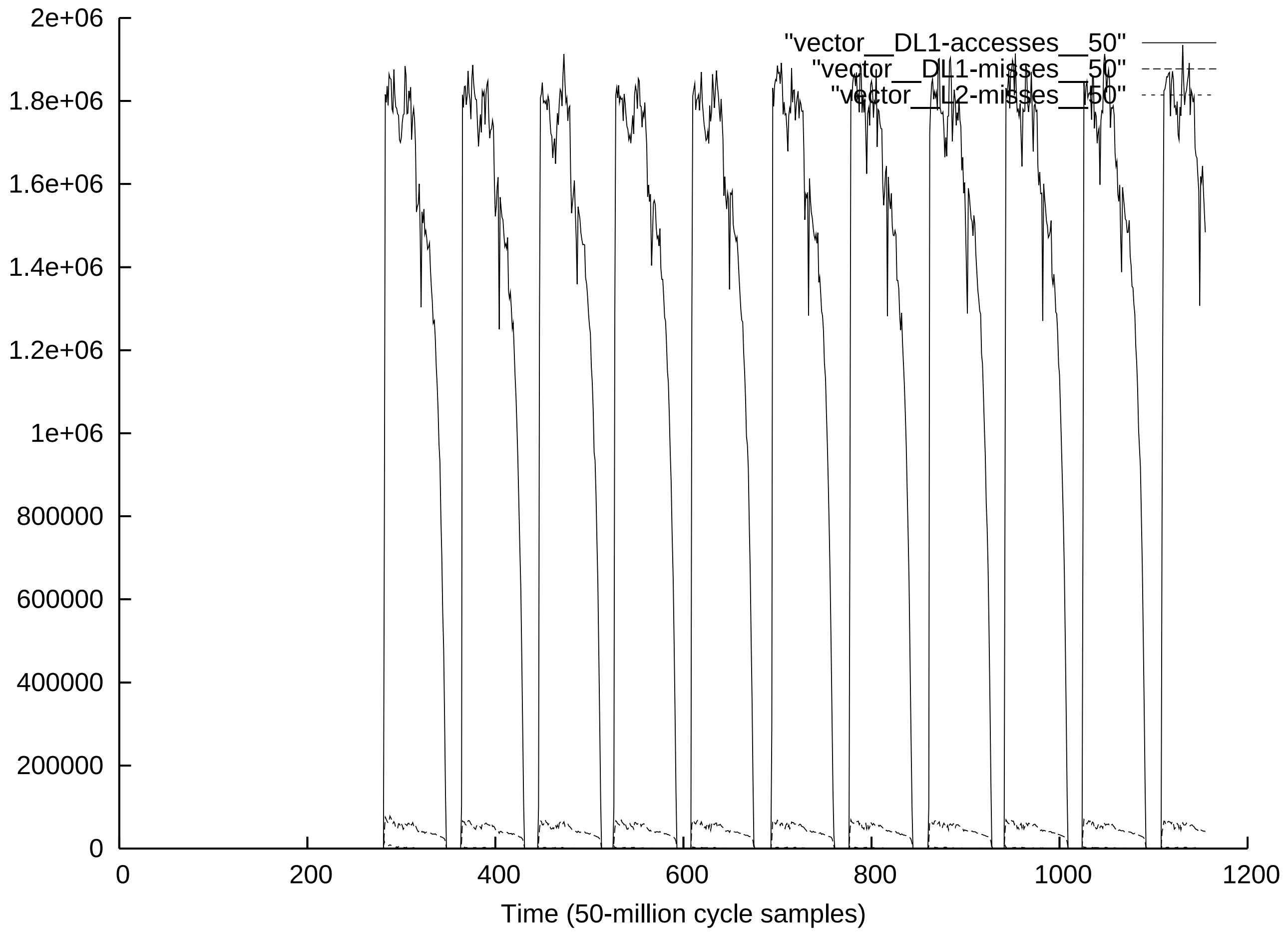


Figure 30: 188.ammp – vector

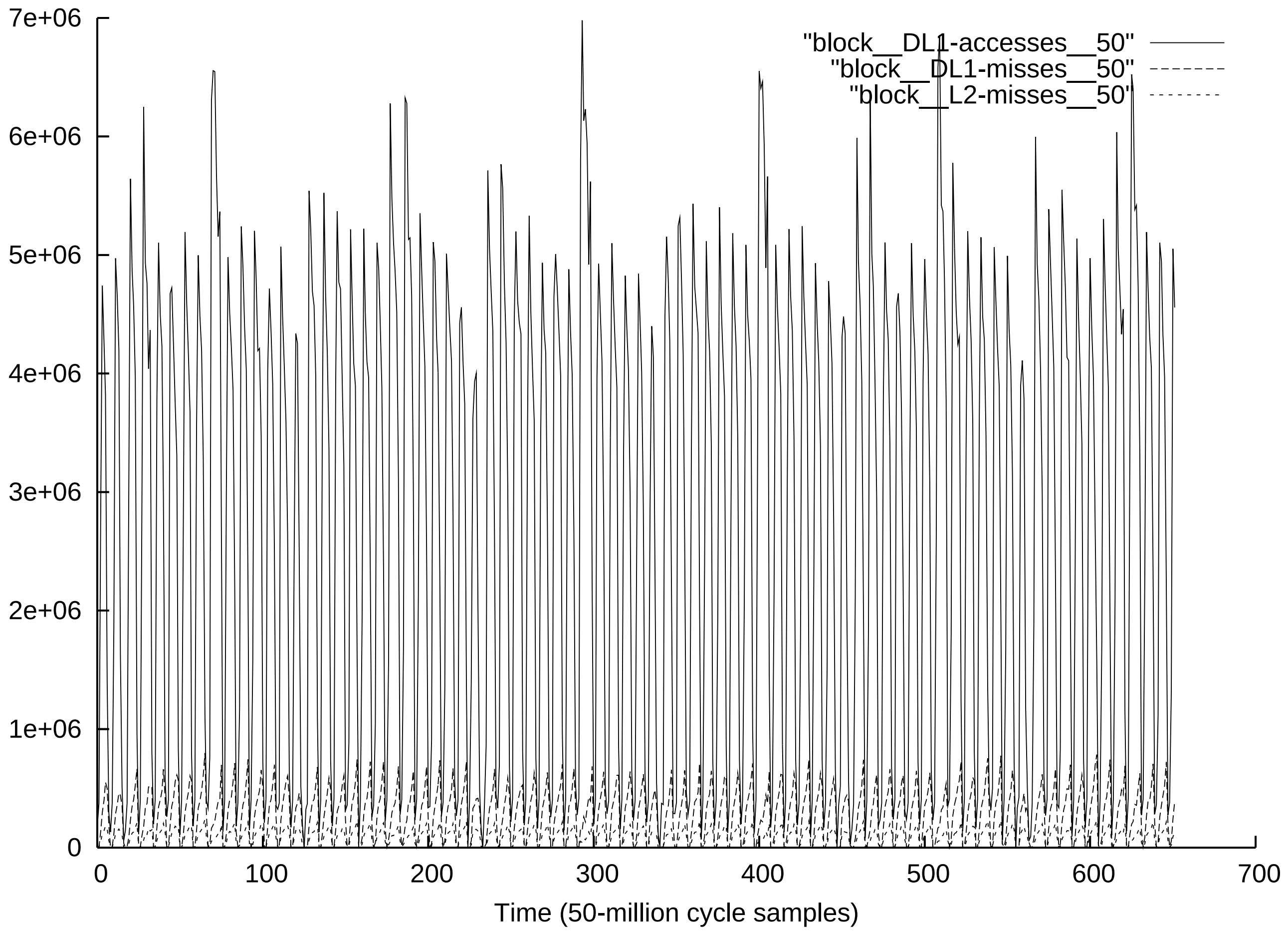


Figure 31: 256.bzip2 – block

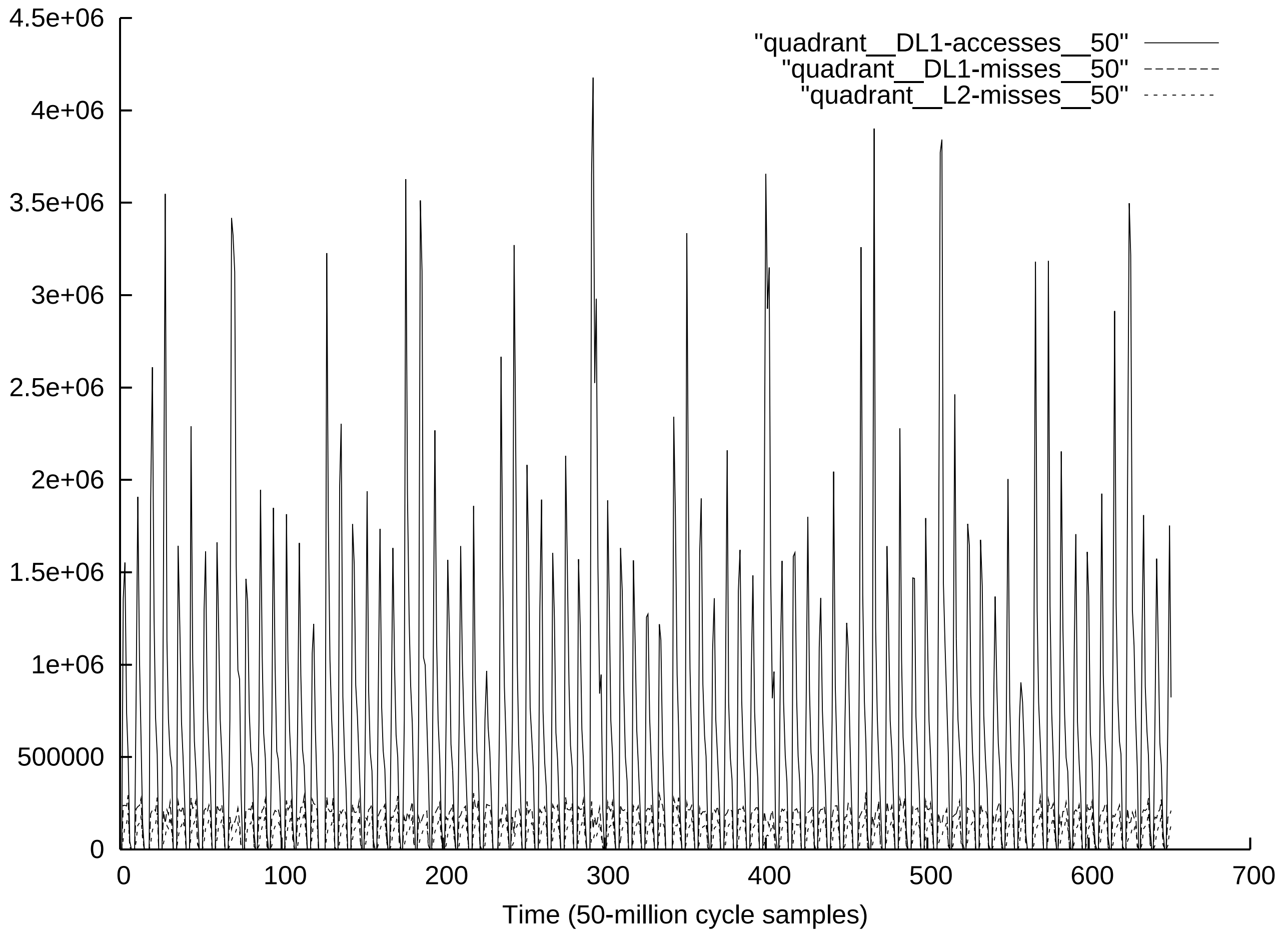


Figure 32: 256.bzip2 – quadrant

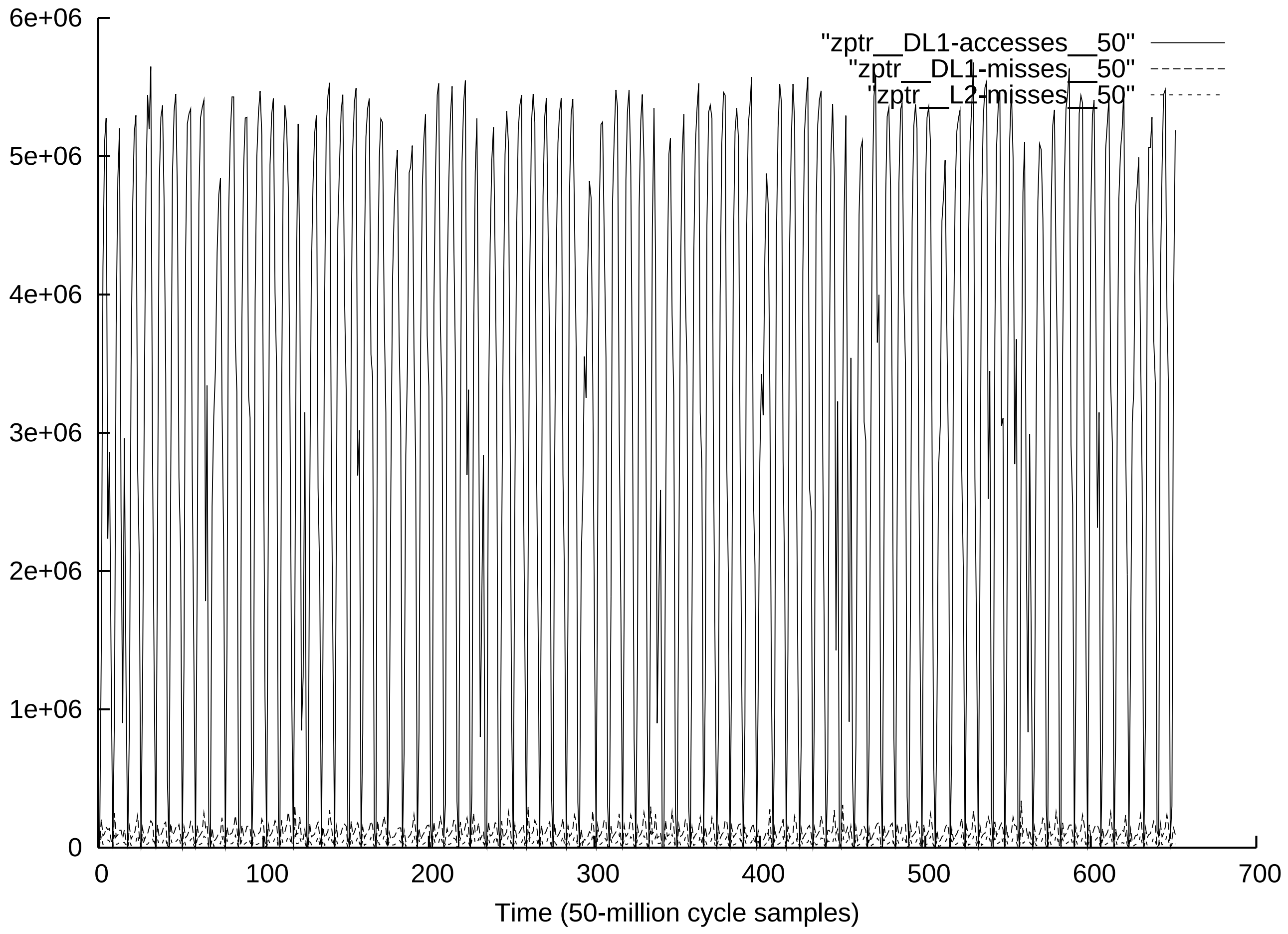


Figure 33: 256.bzip2 – zptr

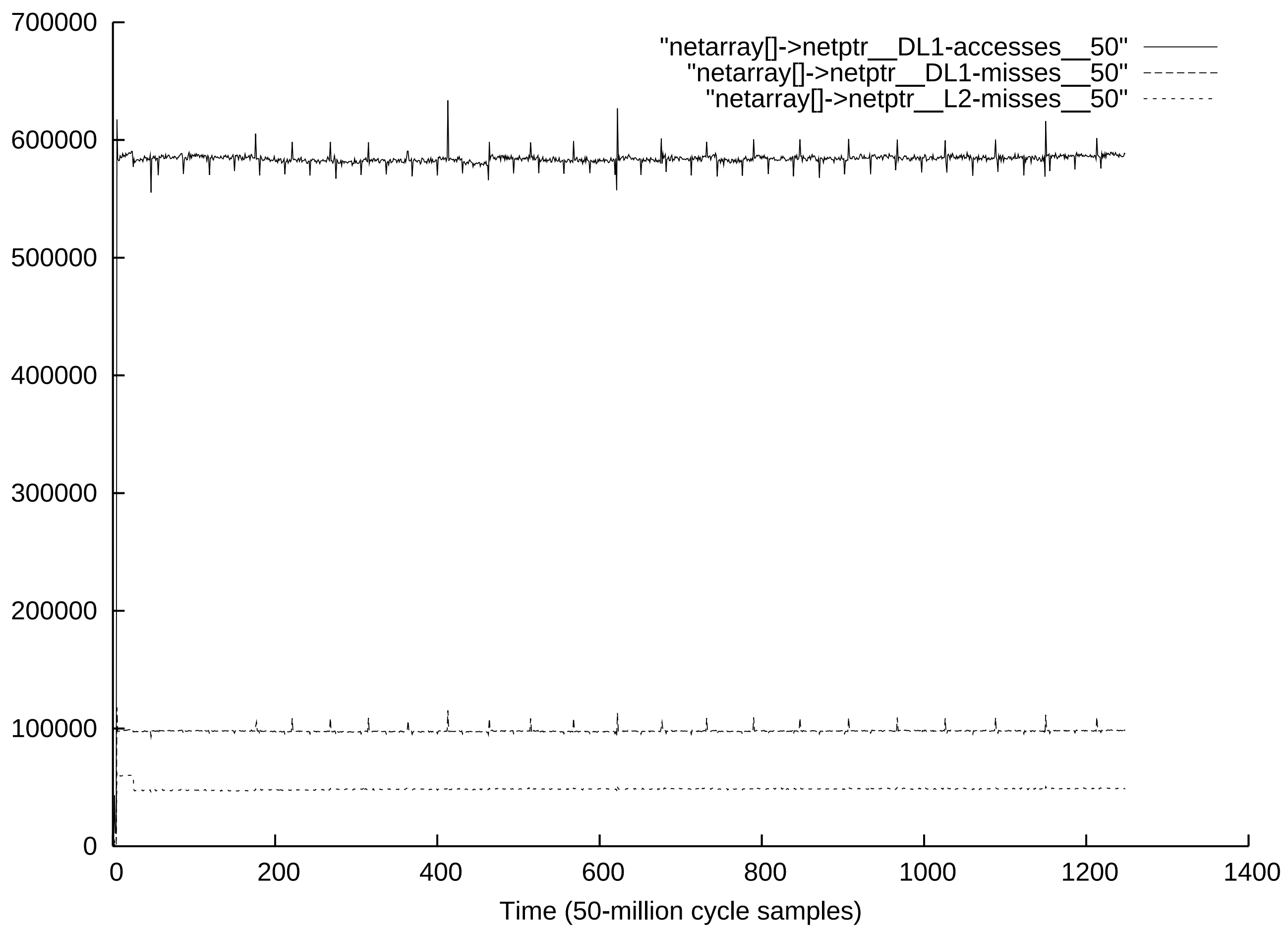


Figure 34: `300.twolf` – `netarray[]`→`netptr`

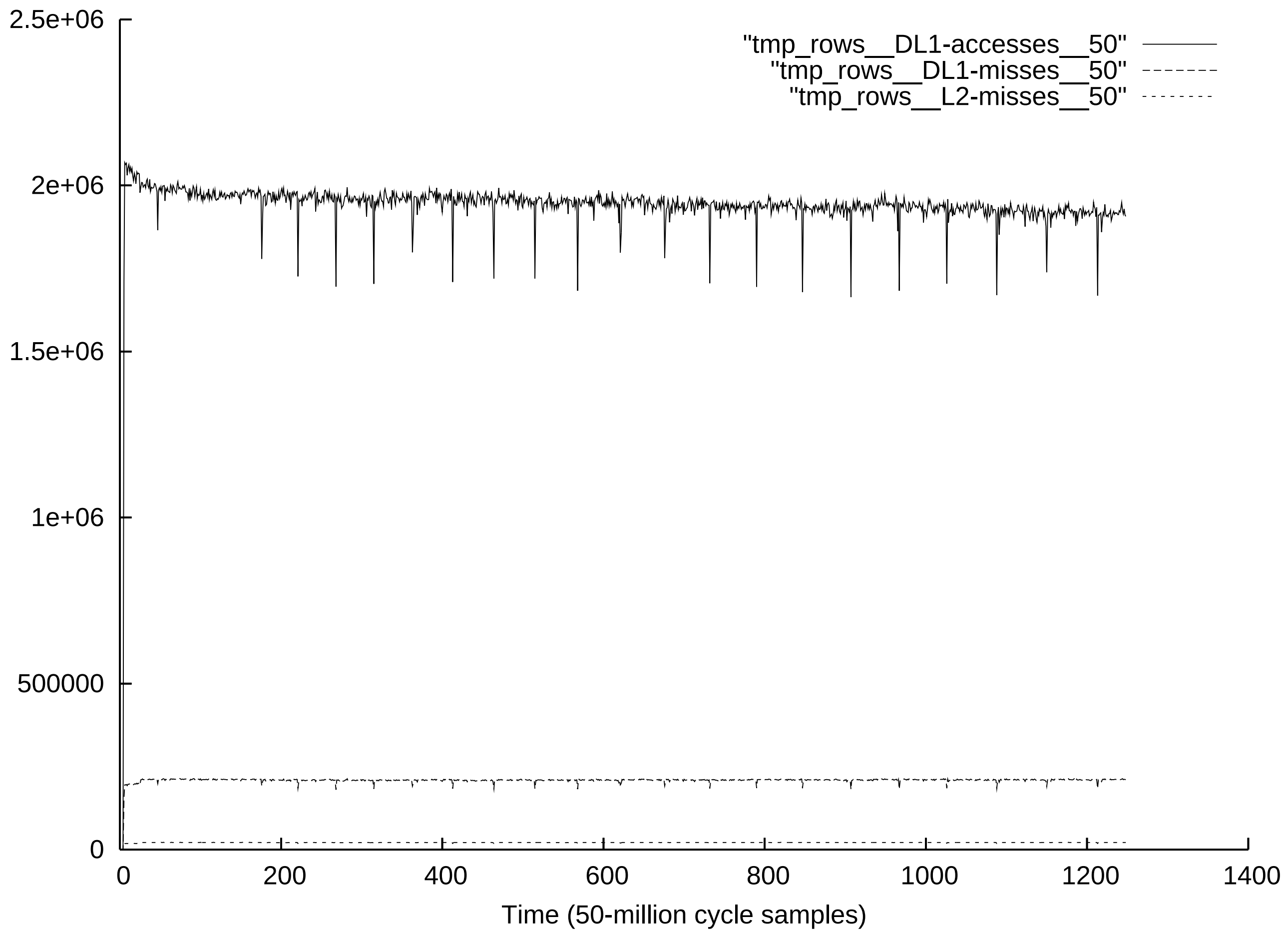


Figure 35: `300.twolf` – `tmp_rows[]`

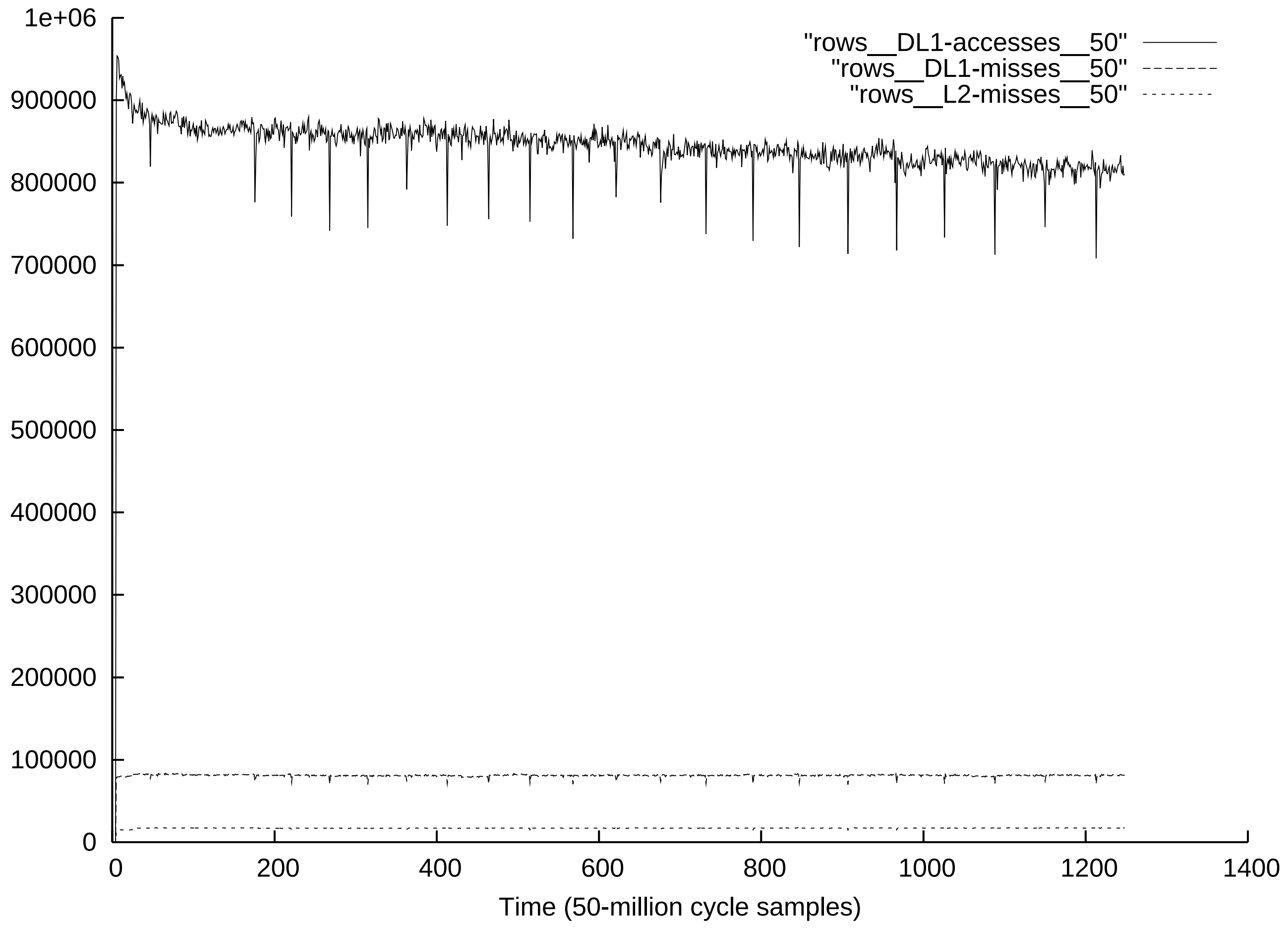


Figure 36: 300.twolf – `rows[]`

Bibliography

[1] S. G. Abraham, R. A. Sugumar, D. Windheiser, B. R. Rau, and R. Gupta. Predictability of load/store instruction latencies. In *Proceedings of the 28th International Symposium on Microarchitecture*, pages 139–152, Austin, TX, December 1993.

[2] V. Agarwal, M.S. Hrishikesh, S. W. Keckler, and Doug Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 248–259, Vancouver, BC, June 2000.

[3] Hassan Al-Sukhni, Ian Bratt, and Daniel A. Connors. Compiler-directed content-aware prefetching for dynamic data structures. In *The 2003 International Conference on Parallel Architectures and Compilation Techniques*, page 91, 2003.

[4] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo. The IBM System/360 Model 91: Machine philosophy and instruction-handling. *IBM Journal of Research and Development*, 11(1):8–24, 1967.

[5] Murali Annavaram, Jignesh M. Patel, and Edward S. Davidson. Data prefetching by dependence graph precomputation. In *Proceedings of the*

28th annual international symposium on Computer architecture, pages 52–61, 2001.

[6] Murali Annavaram, Ryan Rakvic, Marzia Polito, Jean-Yves Bouguet, Richard Hankins, and Bob Davies. The fuzzy correlation between code and performance predictability. In *Proceedings of the 37th Annual International Symposium on Microarchitecture*, pages 93–104, 2004.

[7] Bruno Blanchet. Escape analysis: correctness proof, implementation and experimental results. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–37, 1998.

[8] Keith Boland and Apostolos Dollas. Predicting and precluding problems with memory latency. *IEEE Micro*, 14(4):59–67, 1994.

[9] D. Burger and T. M. Austin. The simplescalar tool set version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin, June 1997.

[10] D. Burger, J. R. Goodman, and A. Kägi. The declining effectiveness of dynamic caching for general-puropose microprocessors. Technical Report 1216, Dept. of Computer Science, University of Wisconsin at Madison, January 1995.

[11] D. Burger, S. Kaxiras, and J. R. Goodman. DataScalar architectures.

In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 338–349, Denver, CO, June 1997.

[12] B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in java controller. In *The 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 280–291, Barcelona, Spain, September 2001.

[13] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, Santa Clara, CA, April 1991.

[14] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a smarter memory controller. In *Fifth International Symposium on High Performance Computer Architecture*, Orlando, FL, January 1999.

[15] T. I. Chappell, B. A. Chappell, S. E. Schuster, J. W. Allan, S. P. Klepner, R. V. Joshi, and R. L. Franch. A 2-ns cycle, 3.8-ns access 512-kb CMOS ECL SRAM with a fully pipelined architecture. *IEEE Journal of Solid-State Circuits*, 26(11):1,577–1,585, November 1991.

[16] M. Charney and A. Reeves. Generalized correlation-based hardware prefetching. Technical Report EE CEG 95-1, Cornell University, Feb 1995.

[17] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 296–310, White Plains, NY, June 1990.

[18] T. Chen and J. Baer. Reducing memory latency via non-blocking and prefetching caches. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 51–61, Boston, MA, October 1992.

[19] T. Chen and J. Baer. Effective hardware based data prefetching. *IEEE Transactions on Computers*, 44(5):609–623, May 1995.

[20] Trishul M. Chilimbi and Martin Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proceeding of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, 2002.

[21] J. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the Twentieth Annual ACM Symposium on the Principles of Programming Languages*, pages 232–245, Charleston, SC, January 1993.

[22] Seungryul Choi, Nicholas Kohout, Sumit Pamnani, Dongkeun Kim, and Donald Yeung. A general framework for prefetch scheduling in linked data structures and its application to multi-chain prefetching. *ACM Trans. Comput. Syst.*, 22(2):214–280, 2004.

[23] Robert Cooksey, Stephan Jourdan, and Dirk Grunwald. A stateless, content-directed data prefetching mechanism. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 279–290, 2002.

[24] M. Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 35–46, Vancouver, BC, June 2000.

[25] Rajagopalan Desikan, Doug Burger, and Stephen W. Keckler. Measuring experimental error in microprocessor simulation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 266–277, July 2001.

[26] Alain Deutsch. On the complexity of escape analysis. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 358–371, 1997.

[27] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural Points-to analysis in the presence of function pointers. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 242–256, June 1994.

[28] Brian Fields, Shai Rubin, and Rastislav Bod&#237;k. Focusing processor policies via critical-path prediction. In *Proceedings of the 28th*

annual international symposium on Computer architecture, pages 74–85, 2001.

[29] D. Grove and L. Torczon. Interprocedural constant propagation: A study of jump function implementations. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 90–99, Albuquerque, NM, June 1993.

[30] Samuel Z. Guyer, Daniel A. Jiménez, and Calvin Lin. The C-Breeze compiler infrastructure. Technical Report TR 01-43, Dept. of Computer Sciences, University of Texas at Austin, November 2001.

[31] Ilkka J. Haikala and Petri H. Kutvonen. Split cache organizations. In *Performance '84: Proceedings of the Tenth International Symposium on Computer Performance Modelling, Measurement and Evaluation*, pages 459–472. North-Holland, 1985.

[32] J. Hennessy and D. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, San Mateo, CA, 1995.

[33] M. D. Hill. A case for direct-mapped caches. *IEEE Computer*, 21(12):25–40, December 1988.

[34] M. Hind, M. Burke, P. Carini, and J. Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems*, 21(4):848–894, July 1999.

[35] Michael Hind, Michael Burke, Paul Carini, and Sam Midkiff. An empirical study of precise interprocedural array analysis. *Sci. Program.*, 3(3):255–271, 1994.

[36] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, 1990.

[37] Ibrahim Hur and Calvin Lin. Adaptive history-based memory schedulers. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 343–354, 2004.

[38] Ibrahim Hur and Calvin Lin. Memory prefetching using adaptive stream detection. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 397–408, 2006.

[39] W. Hwu and Y. N. Patt. HPSm, a high performance restricted data flow architecture having minimal functionality. In *Proceedings of the 13th annual international symposium on Computer architecture*, pages 297–306, Los Alamitos, CA, USA, 1986.

[40] Sorin Iacobovici, Lawrence Spracklen, Sudarshan Kadambi, Yuan Chou, and Santosh G. Abraham. Effective stream-based and execution-based data prefetching. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pages 1–11, 2004.

[41] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. In *Proceedings of the 24th annual international symposium on Computer architecture*, pages 252–263, 1997.

[42] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 364–373, Seattle, WA, June 1990.

[43] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache prefetch buffers. In *25 Years ISCA: Retrospectives and Reprints*, pages 388–397, 1998.

[44] Spiros Kalogeropulos, Mahadevan Rajagopalan, Vikram Rao, Yonghong Song, and Partha Tirumalai. Processor aware anticipatory prefetching in loops. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, page 106, 2004.

[45] M. Karlsson, F. Dahlgren, and P. Stenstrom. A prefetching technique for irregular accesses to linked data structures. Toulouse, France, January 2000.

[46] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 211–222, 2002.

[47] Dongkeun Kim and Donald Yeung. A study of source-level compiler algorithms for automatic construction of pre-execution code. *ACM Transactions on Computer Systems*, 22(3), 2004.

[48] Kiyoshi Kurihara, David Chaiken, and Anant Agarwal. Latency tolerance through multi-threading in large-scale multiprocessors. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, pages 91–101, 1991.

[49] An-Chow Lai, Cem Fide, and Babak Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *Proceedings of the 28th annual international symposium on Computer architecture*, pages 144–154, 2001.

[50] Loren Larsen and Mary Jean Harrold. Slicing object-oriented software. In *ICSE '96: Proceedings of the 18th international conference on Software engineering*, pages 495–505, 1996.

[51] Jeremy Lau, Jack Sampson, Erez Perelman, Greg Hamerly, and Brad Calder. The strong correlation between code signatures and performance. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, March 2005.

[52] Jeremy Lau, Stefan Schoenmackers, and Brad Calder. Structures for phase classification. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, March 2004.

[53] A. R. Lebeck and D. A. Wood. Cache profiling and the SPEC benchmarks: A case study. *IEEE Computer*, 27:15–26, October 1994.

[54] 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 *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 117–128, 2002.

[55] W. Lin, S. K. Reinhardt, and D. Burger. Reducing DRAM latencies with an integrated memory hierarchy design. In *International Conference on High-Performance Computer Architecture*, pages 301–312, Monterrey, Mexico, January 2001.

[56] M. H. Lipasti, W. J. Schmidt, S. R. Kunkel, and R. R. Roediger. SPAID: Software prefetching in pointer- and call-intensive environments. In *Proceedings of the 28th Annual IEEE/ACM International Symposium on Microachitecture*, 1995.

[57] Michael Litzkow, Miron Livny, and Matt Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104–111, June 1988.

[58] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donnell, and J. C. Ruttenberg. The multiflow trace scheduling compiler. *The Journal of Supercomputing*, 7:51–143, 1993.

[59] C. Luk and T. C. Mowry. Cooperative prefetching: Compiler and hardware support for effective instruction prefetching in modern processors. In *Proceedings of the 31st International Symposium on Microarchitecture*, Dallas, TX, December 1998.

[60] M. Martonosi, A. Gupta, and T. E. Anderson. MemSpy: Analyzing memory system bottlenecks in programs. In *Proceedings of the ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems*, pages 1–12, Newport, RI, June 1992.

[61] M. Martonosi, A. Gupta, and T. E. Anderson. Effectiveness of trace sampling for performance debugging tools. In *Proceedings of the ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems*, pages 248–259, Santa Clara, CA, May 1993.

[62] K. S. McKinley, J. Burrill, M. Bond, D. Burger, B. Cahoon, J. Gibson, J. E. B. Moss, A. Smith, Z. Wang, and C. Weems. The scale compiler. Technical report, University of Massachussetts at Amherst, 2005. http://ali-www.cs.umass.edu/∼scale/.

[63] K. S. McKinley and O. Temam. A quantitative analysis of loop nest locality. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 94–104, Cambridge, MA, October 1996.

[64] K. S. McKinley and O. Temam. Quantifying loop nest locality using

SPEC'95 and the Perfect benchmarks. *ACM Transactions on Computer Systems*, 17(4):288–336, November 1999.

[65] Soo-Mook Moon and Kemal Ebcioglu. An efficient resource-constrained global scheduling technique for superscalar and vliw processors. In *MICRO 25: Proceedings of the 25th annual international symposium on Microarchitecture*, pages 55–71, Los Alamitos, CA, USA, 1992.

[66] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, page 129, 2003.

[67] C. R. Myers. Software systems as complex networks: structure, function, and evolvability of software collaboration graphs. *Physical Review E*, 68:046116–1–046116–15, 2003.

[68] Priya Nagpurkar, Michael Hind, Chandra Krintz, Peter Sweeney, and V.T. Rajan. Online phase detection algorithms. In *Proceedings of the 4th annual international symposium on code generation and optimization*, March 2006.

[69] V. Pai and S. Adve. Comparing and combining read miss clustering and software prefetching. In *The 2001 International Conference on Parallel Architectures and Compilation Techniques*, 2001.

[70] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. The impact of instruction-level parallelism on multiprocessor performance and simulation methodology. In *Proceedings of the 3rd International Symposium on High Performance Computer Architecture*, pages 72–83, February 1997.

[71] S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the 21th International Symposium on Computer Architecture*, pages 24–33, Chicago, IL, April 1994.

[72] D. L. Parnas, P. C. Clements, and D. M. Weiss. The modular structure of complex systems. In *ICSE '84: Proceedings of the 7th international conference on Software engineering*, pages 408–417, Piscataway, NJ, USA, 1984.

[73] Steven A. Przybylski. *Cache and memory hierarchy design: a performance-directed approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[74] Moinuddin K. Qureshi, David Thompson, and Yale N. Patt. The v-way cache: Demand based associativity via global replacement. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 544–555, 2005.

[75] A. G. Reinig. Alias analysis in the DEC C and C++ compilers. *Digital Technical Journal*, 10(1):48–57, 1999.

[76] A. Roth and G. Sohi. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the 26th International Symposium on Computer Architecture*, Atlanta, GA, May 1999.

[77] Amir Roth, Andreas Moshovos, and Gurindar S. Sohi. Dependence based prefetching for linked data structures. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115–126, 1998.

[78] Shai Rubin, Rastislav Bodik, and Trishul M. Chilimbi. An efficient profile-analysis framework for data-layout optimizations. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2002.

[79] Radu Rugina and Martin Rinard. Automatic parallelization of divide and conquer algorithms. In *PPoPP '99: Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 72–83, 1999.

[80] A. Seznec. A case for two-way skewed associative caches. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 169–178, San Diego, CA, May 1993.

[81] Andr&#233; Seznec. A case for two-way skewed-associative caches. *SIGARCH Comput. Archit. News*, 21(2):169–178, 1993.

[82] Xipeng Shen, Yutao Zhong, and Chen Ding. Locality phase prediction. In *Proceedings of the Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 165–176, 2004.

[83] Timothy Sherwood, Erez Perelman, and Brad Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *The 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, September 2001.

[84] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, October 2002.

[85] Timothy Sherwood, Suleyman Sair, and Brad Calder. Predictor-directed stream buffers. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 42–53, 2000.

[86] Timothy Sherwood, Suleyman Sair, and Brad Calder. Phase tracking and prediction. In *Proceedings of the 30th International Symposium of Computer Architecture*, pages 336–347, June 2003.

[87] A. J. Smith. Cache memories. *Computing Surveys*, 14(3):473–530, September 1982.

[88] A. J. Smith. Bibliography and readings on CPU cache memories and related topics. *Computer Architecture News*, 14(1):22–42, January 1986.

[89] A. J. Smith. Second bibliography on cache memories. *Computer Architecture News*, 19(4):154–182, June 1991.

[90] James E. Smith. Decoupled access/execute computer architectures. *ACM Trans. Comput. Syst.*, 2(4):289–308, 1984.

[91] James E. Smith and Andrew R. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Trans. Comput.*, 37(5):562–573, 1988.

[92] S. Srinivasan, R. Ju, A. R. Lebeck, and C. Wilkerson. Locality vs. criticality. In *Proceedings of the 28th International Symposium on Computer Architecture*, pages 132–144, June 2001.

[93] Karthik Sundaramoorthy, Zach Purser, and Eric Rotenburg. Slipstream processors: improving both performance and fault tolerance. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257–268, 2000.

[94] Michael Bedford Taylor, Walter Lee, Jason Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jason Kim, James Psota, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ilp and streams.

In *Proceedings of the 31st annual international symposium on Computer architecture*, page 2, 2004.

[95] Teck Bok Tok, Samuel Z. Guyer, and Calvin Lin. Efficient flow-sensitive interprocedural data-flow analysis in the presence of pointers. In *Proceedings of the 15th International Conference on Compiler Construction (CC 2006)*, pages 17–31, 2006.

[96] E. van der Deijl, G. Kanbier, O. Temam, and E. Granston. A cache visualization tool. *IEEE Computer*, 30:71–78, July 1997.

[97] S. P. VanderWiel and D. J. Lilja. A compiler-assisted data prefetch controller. In *Proceedings of International Conference on Computer Design*, pages 372–377, October 1999.

[98] Z. Wang, D. Burger, K. S. McKinley, S. Reinhardt, and C. C. Weems. Guided region prefetching: A cooperative hardware/software approach. In *Proceedings of the 30th International Symposium on Computer Architecture*, pages 388–398, San Diego, CA, June 2003.

[99] Zhenlin Wang, Doug Burger, Steven K. Reinhardt, Kathryn S. McKinley, and Charles C Weems. Guided region prefetching: A cooperative hardware/software approach. In *Proceedings of the 30th International Symposium on Computer Architecture*, June 2003.

[100] Mark Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th*

international conference on Software engineering, pages 439–449, Piscataway, NJ, USA, 1981.

[101] P. R. Wilson, M. S. Lam, and T. G. Moher. Effective static-graph reorganization to improve locality in garbage-collected systems. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 177–191, Toronto, Canada, June 1991.

[102] Chia-Lin Yang and Alvin R. Lebeck. Push vs. pull: data movement for linked data structures. In *ICS '00: Proceedings of the 14th international conference on Supercomputing*, pages 176–186, 2000.

[103] Chia-Lin Yang, Alvin R. Lebeck, Hung-Wei Tseng, and Chien-Hao Lee. Tolerating memory latency through push prefetching for pointer-intensive applications. *ACM Trans. Archit. Code Optim.*, 1(4):445–475, 2004.

[104] Weifeng Zhang, Dean M. Tullsen, and Brad Calder. Accelerating and adapting precomputation threads for efficient prefetching. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture*, pages 85–95, 2007.

[105] Huiyang Zhou. Dual-core execution: Building a highly scalable single-thread instruction window. In *The 2005 International Conference on Parallel Architectures and Compilation Techniques*, pages 231–242, 2005.

Vita

Kartik Kandadai Agaram graduated from high school in 1995 out of Kendriya Vidyalaya, Picket, Secunderabad, India. In 1999 he received a Bachelor of Engineering (B.E.) degree in Computer Science and Engineering from Sri Venkateswara College of Engineering, University of Madras, India. He entered the graduate program in the Department of Computer Sciences at the University of Texas at Austin in August 1999, and received a Master of Science (M.S.) degree in 2005.

Permanent address: akkartik@gmail.com

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.