UPTEC IT 11 003
Examensarbete 30 hp
Januari 2011

Low-Overhead Memory Access Sampler

An efficient method for data-locality profiling.

Peter Vestberg

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract
Low-Overhead Memory Access Sampler

Peter Vestberg

There is an ever widening performance gap between processors and main memory, a gap bridged by small intermediate memories, cache memories, storing recently referenced data. A miss in the cache is an expensive operation because it requires data to be fetched from main memory. It is therefore crucial to understand application cache behavior. Caches only work well for applications with good data locality; insufficient data locality leads to poor cache utilization which quickly becomes a major performance bottleneck. Analysing and understanding the cache behavior helps in improving data locality and identifying such bottlenecks.

In this thesis, we study a method for efficiently analysing application cache behavior. We implement the method in a cache analysis tool. The method uses a statistical cache model that only requires a sparse data locality fingerprint as input. The input is based on reuse distances between cache lines. By adjusting architecture-specific parameters, such as cache line size, the tool can output working-set graphs for a wide range of architectures. Readily available hardware performance counters combined with intelligent sampling are used to enable an implementation with low overhead.

We evaluate our cache analysis tool using the SPEC CPU2006 benchmarks and our results show good accuracy and performance. The difference between the cache miss ratio estimated by our tool and a reference tool was nearly always below one percentage point. The run-time overhead was on average 17%. We also do an analysis of the overhead to identify the components of our implementation that are most costly and should be the focus for optimizations.

We propose a number of optimizations that could reduce the overhead further. Phase-guided sampling is proposed as a key optimization where application phase behavior is used to determine when to sample memory references. We also build a prototype implementation of this optimization and the preliminary results were promising.

Handledare: Andreas Sandberg
Ämnesgranskare: Erik Hagersten
Examinator: Anders Jansson
ISSN: 1401-5749, UPTEC IT 11 003
Sponsor: SSF CoDeR-MP

Tryckt av: Reprocentralen ITC

Populärvetenskaplig Sammanfattning

Swedish Summary

Prestandaskillnaden mellan moderna processorer och arbetsminne ökar ständigt. För att minska skillnaden använder man cacheminnen - små mellanliggande minnen som innehåller senast använd data. Om data som inte finns inläst i cachen efterfrågas av processorn blir resultatet en cachemiss. Vid en cachemiss måste data hämtas från arbetsminnet, vilket är en långsam operation. Det är alltså önskvärt att minimera dessa cachemissar för att få bra prestanda. Således är det mycket viktigt att analysera applikationers cachebeteenden. Cacheminnen fungerar endast väl då applikationer har god datalokalitet. Principen om datalokalitet bygger på observationen att nyligen använd data, och närliggande data, troligen kommer att återanvändas inom kort. Otillräcklig datalokalitet leder till bristfälligt utnyttjande av cachen och därmed dålig prestanda. Analys av hur applikationer använder cachen är ett väsentligt verktyg för att förbättra datalokalitet och identifiera prestandaflaskhalsar.

I detta examensarbete studerar vi en metod för att effektivt kunna analysera applikationers cachebeteenden. Vi implementerar metoden på riktig hårdvara i ett verktyg för cacheanalys. Många existerande metoder har en alltför hög påläggskostnad för att kunna analysera vardagliga applikationer som använder realistiska datamängder. I praktiken används istället kraftigt reducerade datamängder för att kunna genomföra cacheanalyserna. Resultatet blir då ofta mer eller mindre inkorrekta analyser med undermålig noggrannhet eftersom icke-representativa data analyserades. Risken är vidare att felaktiga slutsatser dras från analysen. Målet för vår metod och implementation är att ha en så låg påläggskostnad som möjligt med bibehållen noggrannhet.

Den metod vi implementerar använder en modell av cacheminnet som bygger på statistik. Som indata kräver cachemodellen endast en minimal uppskattning av datalokaliteten hos den studerade applikationen. Denna data bygger på s.k. reuse-distanser och insamlas genom att granska strömmen av minnesreferenser som utförs av applikationen. Reuse-distans är ett bra mått på datalokalitet och bestäms genom att notera hur många minnesreferenser som sker innan viss data återanvänds. För att indatan ska vara så liten som möjligt men fortfarande representativ, samplar vi strömmen av minnesreferenser. Cachemodellen använder sannolikhetslära och numeriska metoder för att omvandla reuse-distansinformationen till en cachemisskvot, d.v.s. andelen minnesreferenser som är cachemissar. Metoden är applicerbar på en mängd olika datorarkitekturer. Genom att endast variera ett fåtal arkitekturberoende parametrar, t.ex. cachestorlek, kan cachemisskvoter beräknas för olika arkitekturer. Vi använder prestandaövervakningsfunktioner i hårdvara kombinerat med intelligenta samplingstekniker för att möjliggöra en im-

plementation med mycket låg påläggskostnad.

Vi utvärderar vårt cacheanalysverktyg med avseende på noggrannhet och prestanda. För detta ändamål använder vi SPEC CPU2006, en industristandard för utvärdering innehållande en rad beräknings- och minnesintensiva applikationer. Vi jämför resultaten från vår implementation med en långsam men noggrann referensimplementation. Resultaten är goda; verktygen uppvisar stor noggrannhet och en låg påläggskostnad. Vidare gör vi även en analys av påläggskostnaden för att ta reda på vilka komponenter i vår implementation som är mest kostsamma och har störst optimeringspotential.

Vi föreslår ett antal optimeringar som kan reducera påläggskostnaden ytterligare. En av de mest intressanta optimeringarna som föreslås är en samplingsteknik som bygger på fasbeteenden hos applikationer. Det är ett välkänt faktum att många applikationer har tydliga och återkommande faser. Algoritmer för fasdetektering kan användas som vägledning för att veta när sampling bör ske. Vi implementerar även en prototyp av denna optimering och de preliminära resultaten indikerar att stora prestandavinster är möjliga.

Contents

Acknowledgements **iii**

1 Introduction **2**
1.1 Introduction . . . 2
1.2 Problem Description . . . 3
1.3 Objectives . . . 3
1.4 Thesis Structure . . . 3

2 Background **4**
2.1 Cache Memory Review . . . 4
2.2 Cache Modelling . . . 7
2.2.1 Requirements . . . 7
2.2.2 Techniques . . . 7
2.3 StatCache . . . 8
2.3.1 Reuse Distance . . . 9
2.3.2 Sparse Reuse Distance Sampling . . . 10
2.3.3 Probabilistic Cache Model . . . 10
2.4 Hardware Performance Monitoring . . . 12
2.5 Phase-Guided Sampling . . . 13

3 Implementation **16**
3.1 Prerequisites . . . 16
3.2 Application Supervisor . . . 17
3.3 Sampling Mechanism . . . 17
3.3.1 Processor Skid . . . 19
3.3.2 Resolving the Skid Problem . . . 20
3.4 Watchpoints Mechanism . . . 24
3.5 Counting Memory References . . . 25
3.6 Base Implementation . . . 26

4 Evaluation **28**
4.1 Methodology 28
4.2 Experimental Setup 29
4.3 Accuracy 29
4.4 Performance 33
4.4.1 Run-Time Overhead 33
4.4.2 Overhead Breakdown 34

5 Future Work **40**
5.1 Optimized Sampling 40
5.2 Optimized Cache Line Watchpoints 40
5.3 Phase-Guided Sampling 41

6 Summary and Conclusions **44**

References **46**

Acknowledgements

I would like to thank all people who have been involved in my master thesis in any way. Many thanks to my advisor Andreas Sandberg for great help and guidance. I would also like to express my gratitude to my study colleague and friend Andreas Sembrant for good technical discussions. Last, but not least, I would like to thank my girlfriend Åsa Skogö for great support and infinite patience.

This thesis was funded by SSF CoDeR-MP.

1 Introduction

1.1 Introduction

A key factor explaining the run-time performance of many applications is the processor cache memory behavior. The large gap between processor cycle time and off-chip memory access time infers a significant penalty on application performance, if the cache system is poorly utilized. Therefore, it is vital to analyze and understand cache behavior. Performance could be greatly improved if the cache usage is optimized.

Analysing cache behavior helps in identifying performance bottlenecks in applications. The gained knowledge from a cache analysis can guide developers for optimizing code flow and data structures and can lead to direct code changes. Information gathered from a cache analysis can also be used as important feedback to compilers for static code optimizations [23].

One method for studying how an application utilizes the cache is to examine its memory reference stream. By monitoring memory load and store operations, key performance metrics, such as data locality information, can be captured. The data locality property is strongly related to how the cache will be used. This information serves as input to a probabilistic cache model that can clearly identify the cache behavior of the studied application. The cache model employs probability theory and numerical methods to transform the data locality information into cache miss ratio numbers for a range of architectures.

For a cache analysis tool to be useful, it needs to be accurate and efficient. The tool must have a sufficiently low overhead to enable analysis of real-world, potentially large, data sets from real applications. Working with a reduced data set would result in an unrepresentative profile of the application and consequently an inaccurate analysis.

In the method mentioned above, the difficult task is to capture a fingerprint of the data locality from the memory stream. Monitoring every load and store operation offers high accuracy but would result in an unbearable run-time overhead for real-world applications, rendering the analysis tool very obtrusive and cumbersome. Instead, techniques for sampling the memory reference stream can be used. However, while significantly reducing the run-time overhead, the problem of selecting representative samples arises. Failure to do so would directly affect the accuracy of the analysis. Implementing a fair sampling technique is a non-trivial problem due to hardware and software obstacles.

An accurate cache analysis tool with minimal overhead is feasible by leveraging readily available hardware performance counters and intelligent sampling.

1.2 Problem Description

In this master thesis, we implement a cache analysis tool based on the cache modeling method outlined above. The goal is to implement an efficient sampling technique for capturing representative performance data, i.e. data locality information, from the memory reference stream of an analyzed application. The implementation should be running on real hardware, have high accuracy and very low overhead.

1.3 Objectives

The primary goals of this thesis are:

- Implement a cache analysis tool. More specifically, a sampling technique for capturing data locality information from a memory reference stream that is then fed into a probabilistic cache model.
- Focusing on performance to enable cache analysis of real applications using real data sets.
- Evaluate the cache analysis tool.
- Study how to take advantage of phase behavior in applications to further reduce overhead.

1.4 Thesis Structure

This thesis is outline as follows. Chapter 2 gives some the background theory of cache memory and techniques for cache modeling. Details of the cache modeling chosen for this thesis are discussed in depth. The chapter is ended with a brief discussion on hardware performance monitoring and phase-guided sampling.

Chapter 3 gives a thorough walk-trough of the implementation of the cache analysis tool. Required mechanisms, potential problems, obstacles and technical details are discussed.

In Chapter 4, the cache analysis tool is evaluated for accuracy and performance, using a set of real-world benchmarks. Cache miss ratio graphs produced from our tool are compared with reference graphs. Numbers for the run-time overhead are presented and a breakdown of the overhead causes are done.

Future work and possible optimizations are discussed in Chapter 5. Summary and concluding remarks are in Chapter 6.

2 Background

2.1 Cache Memory Review

Accessing data in main memory is a very expensive operation. The processing speed of modern processors is extremely high compared to speed of main memory. Typically, an arithmetic instruction using only internal registers takes very few cycles to complete, while a memory instruction accessing main memory might require hundreds of cycles. This large speed difference is a major performance bottleneck since the processor essentially must stall the execution while waiting for data to arrive from memory. The large gap between processor and main memory speeds needs to be bridged to achieve high performance. This is accomplished by intermediate *cache memory*. The following section will briefly review the cache memory.

The cache memory is located between the CPU and the main memory. It is either located on the processor chip or on a separate module. Cache memory is characterized by small storage capacity and low access latency. The access time to cache memories are close to the cycle time of the CPU. Computer systems often have multiple cache memories organized in a hierarchy, with the smallest and fastest cache closest to the CPU. The first level in hierarchy, closest to the CPU is referred to as the *L1 cache*, the second-closest level is referred to as the *L2 cache* and so on. Processors and cores can also share caches, for example, the L3 cache in Figure 2.1.

When the CPU requests the contents of some memory location, the cache is checked for the requested data. If the data is present in the cache, it can be delivered to the CPU directly from the cache. This is referred to as a *cache hit*. If the data is not present in the cache, it must first be

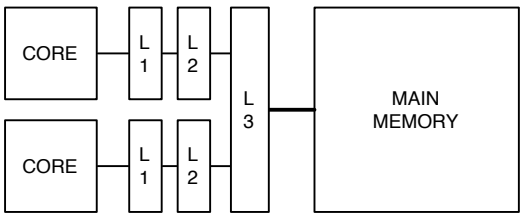


Figure 2.1: Example of a cache hierarchy. A dual core system where each core has private L1 and L2 caches and a shared L3 cache.

fetched from the main memory and loaded into the cache before it can be delivered to the CPU. This is referred to as a *cache miss*. When data is loaded into the cache, some other data in the cache must be replaced, assuming that the cache is full. What data to replace is determined by a *replacement policy*, which is discussed below. This is the basic operation of the cache.

Programs frequently access the same or related data in loops and other iterative constructs. Cache memory should therefore store data that is frequently used by the CPU. Caches are designed with the *principle of locality* in mind. Data locality comes in two flavors, *temporal* and *spatial* locality. Temporal locality means that if some data is referenced, it is likely that the same data is referenced again soon. Temporal locality makes up the foundation for caches. Spatial locality means that data located close to some referenced data is also likely to be referenced. An example of spatial locality would be a simple array traversal. When a cache miss occurs and data must be loaded into the cache from main memory, not just the referenced data is loaded, but a whole block of nearby data. This optimizes for spatial locality. The size of the data that is loaded is referred to as the *line size*. The loaded data and its size are also commonly referred to as *cache block* and *block size*. Cache memory and their design for data locality creates the illusion of the main memory being faster than it really is.

Cache memory is divided into data blocks called *cache lines*. A common cache line size is 64 bytes. An algorithm is needed to map main memory blocks into cache lines. There are three methods for doing so: *direct*, *fully associative* and *set associative* mapping.

Direct mapping maps each block of main memory to one specific cache line. Main memory blocks are typically mapped to cache lines via the equation *cache line = memory addr % #cache lines*. A *direct mapped cache* is simple and inexpensive but might suffer from large miss ratios if two frequently accessed blocks of memory are mapped to the same cache line.

Fully Associative mapping is the opposite of direct mapping. A memory block from main memory can be mapped to any cache line. A cache with this kind of mapping is called a *fully associative cache*. This design allows a greater flexibility when deciding which data in the cache to replace when new data is read into the cache. The downside is a rather complex circuitry. A fully associative cache has an increased (worse) hit time compared a direct-mapped cache, but instead offers decreased (better) miss rates. Consequently, this mapping is best when the miss penalty is very high. An example is the Translation lookaside buffer (TLB). The TLB is a specialized CPU cache that is used for translating virtual memory addresses to physical memory addresses. A miss in the TLB is expensive since it requires a *page table walk*, where the contents of multiple memory locations must be read in order to complete the memory address translation. Therefore, TLBs often have a high or full associativity.

Set associative mapping is a trade-off between direct and associative mapping. The cache is divided into a number of *sets* where each set contains a number of cache lines. A cache memory with n cache lines per set is referred to as a *n-way associative* cache. A main memory block is mapped to one specific set, but to any cache line within that set. This organization can significantly improve the cache hit ratio compared to direct mapping.

Figure 2.2 gives an example of a set associative cache memory. This is the most common cache organization.

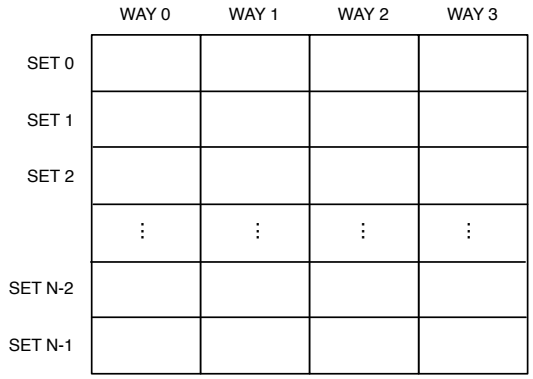


Figure 2.2: A 4-way set associative cache memory with N sets. Assuming 64B cache lines and $N = 32$, the cache size would then be 8kB.

When new data is read into a direct-mapped cache, there is no option of which cache line in the cache to replace or *evict*. For associative and set-associative caches however, there is a choice. A *replacement policy* is used to determine which cache line to throw out from the cache. The replacement policy must be implemented in hardware for speed and is tuned for maximizing the cache hit ratio. Common replacement policies include *Least Recently Used* (LRU), *First-In-First-Out* (FIFO), *Least Frequently Used* (LFU) and *random*.

With LRU policy, the cache line that is least recently used in the set is selected for eviction. This is known as the most effective replacement policy. With FIFO policy, the cache line that has been in the set for the longest time is evicted. The LFU policy replaces the cache line with fewest references to it. The simplest algorithm is the random replacement policy which picks a cache line at random. The usage-based policies are more complex hardware-wise, but usually offer slightly better performance.

When the CPU reads or writes data that is not present in the cache, a cache miss occurs. Data must then be fetched from the main memory with a much larger latency. Cache misses are categorized into three types: *compulsory*, *capacity* and *conflict* misses [9].

Compulsory misses are caused by the first reference to the data. These misses are unavoidable with regards to cache configuration, i.e. cache size, associativity and replacement policy. Compulsory misses are also called *cold misses*. The number of cold misses can be reduced by hardware prefetching. Increasing the block size is a kind of prefetching which helps the matter.

Capacity misses are caused by the finite size of the cache and can only be avoided by increasing the cache size.

Conflict misses are misses that could have been avoided and are caused by too low associativity or a non-optimal replacement policy.

The method used for cache analysis in this thesis does not consider compulsory misses (cold misses). However, this will have a negligible effect on the analysis because the number of compulsory misses are usually small.

2.2 Cache Modeling

2.2.1 Requirements

A crucial property of any performance tool is, of course, its accuracy. For a cache behavior analysis tool, efficiency is another vital property. Analysing the cache behavior requires a real-world data set to produce accurate results. Using a reduced data set may lead to incorrect conclusions. Consequently, the tool needs to be efficient enough to run a realistic data set.

Furthermore, a cache performance tool has to be flexible in order to model a wide variety of cache configurations. Most tools are expected to work on multiple generations of architectures from multiple manufacturers. This implies that a tool must not depend on any architecture-specific features and should only rely on features existing on commodity hardware. It is also desired that the tool is as a unobtrusive as possible; it should preferably be able to profile an application with few or no modifications to the system.

A tool for analysing cache behavior is basically required to capture and examine the memory access stream performed by the analyzed application. To perform this while adhering to the properties above, is a very challenging task.

2.2.2 Techniques

There are a number of different techniques for profiling application performance with respect to memory and cache behavior: cache simulation, code instrumentation, sampling techniques, hardware monitoring, compile-time analysis and statistical methods.

A traditional approach is cache simulators that mimic the cache memory. Simulation allows a very detailed and flexible cache analysis, but comes with a major slowdown. Cache simulators might be incorporated in full-system simulators, like Simics [15] and SimOS [21]. Typically, such tools are trace-driven, requiring large or complete memory reference traces of the studied application. Trace-driven simulation gives very accurate results, but is at the same time very inefficient, considering both disk space and analysis time. Using this approach for evaluating realistic workloads would be very impractical.

Simulation-based cache behavior analysis tools may also be driven by code instrumentation for example CacheGrind [18], SIGMA [5] and CProf [12], on source code [16] or machine code levels. These tools are capable of simulating caches with satisfactory detail, but are still limited by their large slowdown. Also, they might not easily capture operating system interaction.

The large run-time overhead of cache simulation often becomes intolerable for long-running applications. To reduce the overhead, different sampling techniques can be used. Two common

techniques are time sampling [25], [11], [4], and set sampling [16]. With time sampling, continuous sub-traces of the complete memory trace are simulated. This technique requires long warm up periods and is therefore best suited for small caches only. The set sampling technique means that only a subset of the sets in a set-associative cache is simulated. The downside with set sampling is usually defective accuracy.

Sampling guided by application phases is another, more recent, sampling technique [20]. This technique uses the fact that many applications experience well-defined and repetitive phases during their execution. Phase-guided sampling is discussed more in detail in Section 2.5. Common for all sampling techniques is the problem of selecting representative samples. This is explored in [19].

Another common approach is to make use of performance monitoring facilities in hardware. Modern processors have built-in hardware counters for a large range of events, including cache misses, branch miss predictions, stall cycles etc. Using these counters presents a very low overhead but has the disadvantage of being rather architecture-specific. Also it can be difficult to measure metrics not directly available in the hardware, such as data locality.

Cache modeling and analysis might also be compiler-driven [22], [3], [7]. This is a technique where the compiler performs static analysis on the source code to profile data locality and cache usage. The big advantage of this technique is that the application to be analyzed never needs to be run. However, it is limited to fairly well-structured source codes and can only work with the static information known at compile-time.

The cache modeling method chosen for this thesis is a probabilistic-based cache model named *StatCache* [1] [2]. It is best described as a hybrid between fast techniques based on hardware performance monitoring and accurate cache simulation techniques. This method was shown to accurately model cache behavior while promising high speed implementations.

2.3 StatCache

This section will go into the details of StatCache. StatCache models a fully-associative cache with random replacement policy. StatCache uses a probabilistic cache model. The input to the model is information about the application data locality. Using probability theory and numerical methods, StatCache can transform the fingerprint into cache miss ratios. The output is the cache miss ratio for a given target architecture. By varying architecture parameters, such as cache size, cache miss ratios for a range of architectures can be generated. From this, a *working-set graph* can be plotted. A working-set graph is simply a plot of the miss ratio as a function of cache size. Figure 2.3 presents an overview of the StatCache model.

The input to the model is essentially a fingerprint of the data locality of the studied application. More specifically, it is a sparse estimation of the *reuse distance* distribution, collected from sampling. This is described in detail in the following sections. The input is captured by examining the memory reference stream of the application.

The StatCache approach enables an efficient implementation with minimal overhead that can profile large and realistic workloads with maintained accuracy.

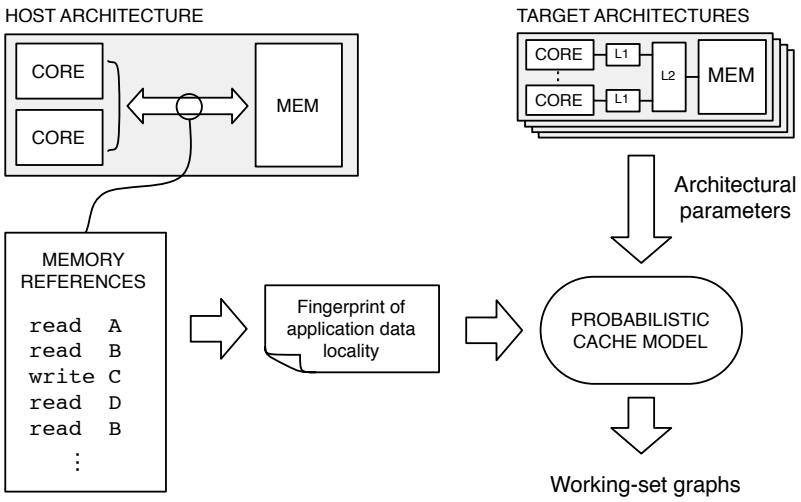


Figure 2.3: Overview of the StatCache cache modeling technique. A fingerprint of application data locality is captured by examining and sampling the memory stream of the analyzed application. The fingerprint is fed into a probabilistic cache model to model cache behavior given architectural parameters.

2.3.1 Reuse Distance

The input to the probabilistic cache model in StatCache is made up of *reuse distances*. Reuse distance is a generic metric that can be used for analysing the data locality property of an application. The reuse distance is defined as the number of memory references between two references to the cache line. More formally:

Definition 1. Let i and j denote two ordered memory references ($i < j$), e.g. i references the memory before j does. Assume that i and j access the same cache line A and that there are no intermediate references to A. Then the *reuse distance* of reference j equals $j - i - 1$. In other words, the number of intermediate memory references between i and j. Reference i and j is called a *reuse pair*.

Reuse distance must not necessarily be defined with respect to cache lines, although this is the most common in the context of cache analysis. In general, any entity in the storage hierarchy can be used.

Figure 2.4 illustrates the concept of reuse distances. It is important to note that the reuse distance is defined as *all* intermediate memory references between two references to the same cache line. Note that, in Figure 2.4, cache line A is referenced twice while measuring the reuse distance for the first reference to cache line B. The reuse distance is still 4 since all intermediate references are counted. Counting only *unique* memory references would give us the *stack distance*, which is a similar metric. The reuse distance is far easier to measure than the stack distance, which is important for an efficient implementation. *StatStack*, a cache model closely

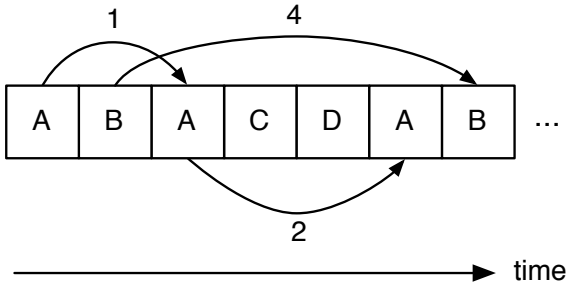


Figure 2.4: The concept of reuse distance. The squares are referenced cache lines. The arcs show cache line reuses and the corresponding reuse distances. For example, the first reference to cache line B is followed by four references to other cache lines before it is referenced again. Thus, the reuse distance equals 4.

related to StatCache, estimates the stack distance for efficient modeling of LRU caches [6].

An important property of the reuse distance metric is the architecture independence. The reuse distance captures the data locality property of an application without requiring any architectural information. This makes the reuse distance an attractive metric for a cache analysis tool where it is desirable to not have to re-run the analysis for every architecture of interest.

2.3.2 Sparse Reuse Distance Sampling

Reuse distance information gives an architecturally independent profile of how the analyzed application uses the memory. However, for a cache analysis tool aiming for good performance, it would be too exhaustive to collect all reuse pairs. This would require a complete memory trace which incurs a major performance penalty. Instead, as already pointed out, the reuse distance information is *sparse* and captured through sampling.

It is sufficient to collect only a subset of all reuse pairs to capture a representative fingerprint of the analyzed application's data locality property [1]. This approach allows for good performance while maintaining accuracy.

The sampling period should preferably be exponentially distributed, so that memory references are sampled with some randomness. This prevents the sampling mechanism to select a static pattern of memory references for sampling. It is of great importance that each memory reference has the same probability of being sampled [19]. If this is not case, the captured fingerprint will be inaccurate and misleading and might lead to incorrect conclusions from the modelled cache behavior.

2.3.3 Probabilistic Cache Model

An application data locality fingerprint based on reuse distances implicitly describes its cache behavior. However, the reuse distance information is hard to interpret as is and would be awkward

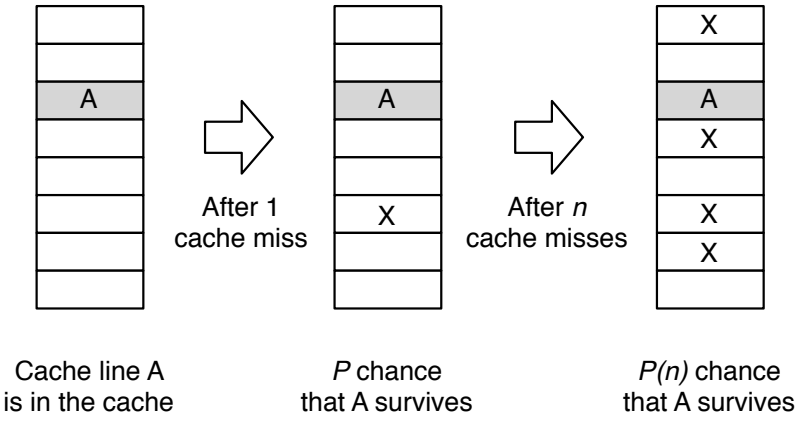


Figure 2.5: Probability of a cache line remaining in the cache after a number of cache misses. Assume a (full) cache with L cache lines. The tick marks denote evicted cache lines.

to directly use for cache behavior analysis. Therefore, the information is fed into a probabilistic cache model which calculates the cache miss ratios for different architectures. This section will explain the mathematics behind this probabilistic cache model.

Assume a fully associative cache with random replacement policy and L cache lines. On a cache miss, a random cache line is selected for replacement or *eviction*. Let the probability that a cache line is evicted be P_e and that it still remains in the cache P:

$$P_e = \frac{1}{L} \tag{2.1}$$

$$P = 1 - P_e = 1 - \frac{1}{L} \tag{2.2}$$

Moreover, let $P(n)$ be the probability that a cache line remains in the cache after n cache misses (replacements):

$$P(n) = P^n = \left(1 - \frac{1}{L}\right)^n \tag{2.3}$$

Figure 2.5 illustrates the probabilities of whether a cache line is evicted or not. The probabilistic cache model in StatCache is based on this basic observation.

Let $f(n)$ be the probability that a cache line has been evicted after n cache misses:

$$f(n) = 1 - P(n) = 1 - \left(1 - \frac{1}{L}\right)^n \tag{2.4}$$

Now assume that the cache miss ratio ratio, M, is constant and known. Also let the reuse distance of a reuse pair be D. The number of cache misses before data is resued can then be estimated as $n = MD$. Consequently, Equation 2.4 can be expressed as a function of the reuse distance:

$$f(MD) = 1 - \left(1 - \frac{1}{L}\right)^{MD} \tag{2.5}$$

Using Equation 2.5, the expected total number of cache misses can be estimated by summing over every memory reference:

$$\text{Total misses} = \sum_{i=0}^{N} f(MD_i) \tag{2.6}$$

where D_i denotes the reuse distance for memory reference i and N the total number of memory references. Furthermore, the total number of cache misses can also be estimated using the assumed constant cache miss ratio M, i.e.:

$$\text{Total misses} = MN \tag{2.7}$$

Consequently, we now have a relationship between the reuse distance and the miss ratio:

$$\sum_{i=0}^{N} f(MD_i) = MN \tag{2.8}$$

Equation 2.8 has only one unknown variable M and can be solved numerically.

In the mathematical reasoning above, we assumed a constant miss ratio. However, this assumption is not valid for all applications. Many applications have a miss ratio that varies during the execution. To overcome this problem, we simply split the execution into small *sampling windows*. The window size is sufficiently small to justify the assumption of a constant miss ratio. Reuse distance information is captured for every window and the window miss ratio is estimated with Equation 2.8. The overall miss ratio for an application is estimated as the arithmetic mean of the miss ratio in every sampling window.

2.4 Hardware Performance Monitoring

In order to efficiently examine the memory reference stream of the analyzed application and to calculate reuse distances, hardware performance monitoring can be used. Modern hardware comes with readily available hardware counters for a range of common events. Examples of hardware counters are the instructions counter, level 1 cache miss counter, floating point multiplication counter and so forth. For our cache analysis tool, we need the counters for memory load and store operations.

The hardware counters can be configured to notify the system after a number of counted events. Basically, a hardware counter can be set to a value close to its maximum value, e.g. $Counter = Counter_{max} - n$, where $Counter_{max}$ is the maximum value the counter can hold and n is the desired number of events before notification. The counter will overflow after n events and an overflow interrupt will be generated by the processor. The interrupt can then be handled appropriately by the operating system.

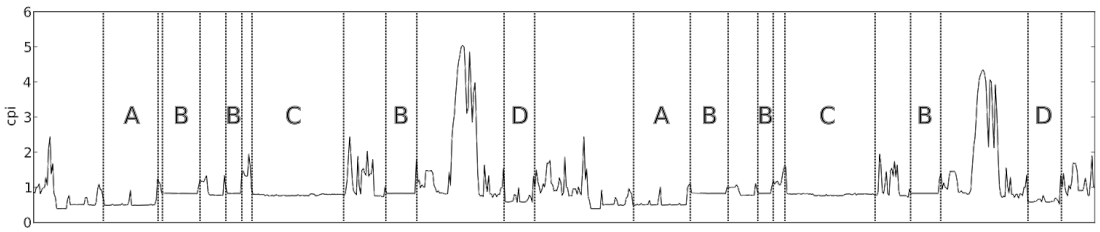


Figure 2.6: Phases in an application. CPI is plotted as a function of time. The figure shows several distinct and re-occurring phases. Note that all phases are not annotated. The figure is borrowed with permission from [24].

A problem with the hardware counters is that the overflow interrupts are deferred. The processor might not stop immediately after the actual overflow occurred, but instead continue execution for some instructions. This problem is known as *skid*. In many cases, it is important to know exactly which instruction that caused the overflow, e.g. the *n*:th event. The presence of skid complicates this. The implications of skid and possible solutions are discussed in depth in Section 3.3.1.

An interesting and relatively new feature included in recent Intel chips is a mechanism called *Precise-Event-Based Sampling* (PEBS). When PEBS is enabled, the register context is saved in a special data buffer immediately when a hardware counter overflows. While PEBS introduces some overhead to the hardware counters, it makes it possible to retrieve the state of the processor when a hardware counter overflowed, despite the skid effect. Using this feature is also discussed in Section 3.3.1.

2.5 Phase-Guided Sampling

Many applications experience well-defined and repetitive phases during their execution. Phase behavior is a well-studied phenomenon. A phase can be defined in many ways. For instance, a phase might be defined by the cycles per instruction (CPI) or the cache miss ratio during a time interval. Another approach is to define a phase by the code that is executed during the interval. This might be done by monitoring the execution of basic blocks. Figure 2.6 shows an example of clearly identified phases.

The performance of the discussed cache modeling technique can potentially be boosted by exploiting the phase behavior of applications. The run-time overhead can be reduced by avoiding redundant sampling. It is likely that a phase has roughly the same behavior, with respect to cache usage, every time it is executed. Sampling the same phase over and over again would only give redundant information. If the phase behavior of the analyzed application can be monitored, this fact can be used to eliminate redundant sampling.

There are very low-overhead algorithms for classifying and predicting phases [24]. By keeping a record over which phases that have been identified and sampled, the sampling mechanism could be completely turned off when previously sampled phases are encountered during the ex-

ecution. Since most phases are re-occurring many times, the sampling mechanism can be turned off for large parts of the execution. This can increase the performance of the cache modeling technique. Phase-guided sampling can be especially valuable when not just the average cache behavior is considered, but the cache behavior over time.

3 Implementation

This chapter describes an efficient and fully functional proof-of-concept implementation of the data acquisition component of StatCache. The implementation targets the Linux operating system and x86 architectures. It is built and tested on a 64-bit Linux 2.6.36 kernel running on an Intel Nehalem machine.

There are two main components to the cache modeling technique. The first component is *online* and captures a data locality fingerprint, in the form of reuse distances, of the analyzed application by examining its memory reference stream. The analyzed application is hereafter referred to as the *target process*. Capturing this information requires monitoring and controlling the target process. This work is performed by a *monitor process*. The efficiency and accuracy of the cache modeling technique is defined by the workings of this online component. Consequently, from an implementation point of view, this is the most interesting and complex component, and thus will be the focus of this chapter. The second component is the probabilistic cache model itself. This component calculates the cache behavior from the application fingerprint in a fraction of a second using numerical methods. Working-set graphs can then be produced. This component is run *offline*. Figure 3.1 shows a high level overview of the cache analysis tool.

3.1 Prerequisites

There are four main mechanisms that are required to implement the online component of our cache analysis tool:

Application supervisor A supervisor that controls and monitors the studied application is required to enable capturing of data locality information.

Sampling mechanism Required to randomly sample memory reference instructions in the target process, and set watchpoints on the cache lines that they reference. The sampling mechanism starts new reuse distance samples.

Watchpoint mechanism Required to detect when a specific cache line is reused. The watchpoints will be referred to as *cache line watchpoints*. The watchpoint mechanisms terminates reuse distance samples.

Memory reference counter Required to measure the length between a reuse pair, i.e. determine the reuse distance, when the watchpoint mechanism has detected a cache line reuse.

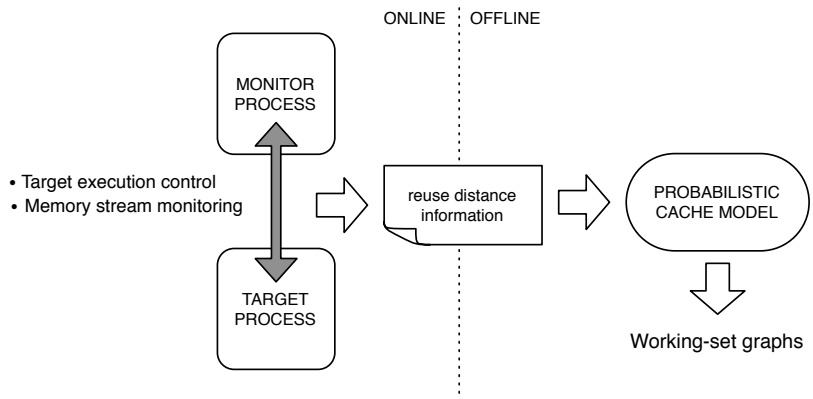


Figure 3.1: Overview of the cache analysis tool implementation. The implementation will focus on the online component. The target process is the studied application which is controlled and monitored by a monitor process.

3.2 Application Supervisor

The monitor process must have a way of supervising the target process in order capture data locality information. More specifically, the monitor must be able to control the execution of the target and inspect its memory and register contents. We are using the traditional *ptrace* debug API to accomplish this.

When the target is being *traced* by the monitor, using ptrace, any interesting events related to the target, such as signals, system calls and changes in execution, will always go through the monitor. In ptrace, most actions are controlled and reported via signals. A signal that is addressed to the target but intercepted and handled by the monitor, is the primary way for ptrace to communicate with the monitor. Such a signal will be referred to as a *pending* signal from here on. For example, when a signal is being delivered to the target, the signal will first be intercepted by the monitor while the target remains stopped (not scheduled). The monitor can then determine what actions to take. For instance, it can read or alter the register context of the target, it can single-step target instructions, suppress the signal or even deliver another signal.

By using ptrace, the monitor has full insight into the target and can control its execution as needed.

3.3 Sampling Mechanism

A key factor for achieving good performance with our cache analysis tool is to use a *sparse* input to the cache model. By selecting only a *small* and *representative* fraction of the memory references (and reuse distances) through sampling, we reduce the input size heavily and thus enable an efficient implementation.

Figure 3.2 illustrates the concept of reuse distance sampling. The sampling mechanism is

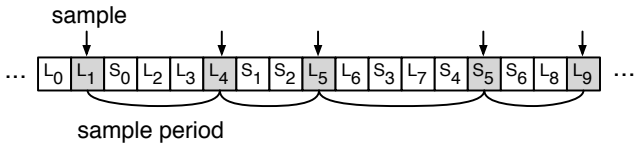


Figure 3.2: Sampling of memory references. The boxes represents a sequential stream of memory references, where *L* represents a load operation and *S* represents a store operation. The shaded boxes are sampled references. The sample period is exponentially distributed. In the example, the target sample period is approximately 3.

implemented using readily available hardware performance counters found in modern processors, see Section 2.4. The relevant counters in our case are the memory load and store operation counters. To use the counters for sampling purposes, we configure them to overflow after a number of loads or stores, i.e. after a sample period. Upon overflow, the processor will generate an interrupt that is handled by the operating system.

One limitation on the implementation system is the lack of a hardware counter counting both load and store operations simultaneously. Attempts were made to configure the load and store counters as a joint counter, but the results were unpredictable and unsatisfactory. To overcome this limitation, we are instead running the sampling mechanism for the load and store counters independently.

We are using the *perf_event* API in the Linux kernel to program the hardware counters. In our implementation, perf_event is configured to deliver a *SIGIO* signal as soon as the overflow interrupt is generated. The signal is set to be delivered to the target process, but is intercepted by the monitor process through the use of ptrace. When the monitor receives the signal, a new reuse distance sample is started.

Starting a new sample includes the following basic steps:

1. **Instruction decoding**. When an overflow interrupt occurs, the target is stopped. We need to decode the (memory) instruction that the target stopped at.

2. **Determine referenced cache line**. Using the decoded instruction and current register contents of the target, compute the memory address and thereafter the corresponding cache line that the instruction references.

3. **Record current time**. Read and record the current number memory references performed by the target so far.

4. **Cache line monitoring**. Start watching the referenced cache line for future reuse. Cache line watchpoints are described in detail later in this chapter.

Instruction decoding is necessary to compute the data address that the sampled instruction references. Since this implementation targets x86 architectures, with complex and variable-length instructions, a full-fledged x86 decoder is necessary. We are using the instruction encoder/decoder *XED*, that is part of the dynamic binary instrumentation framework *Pin* [14].

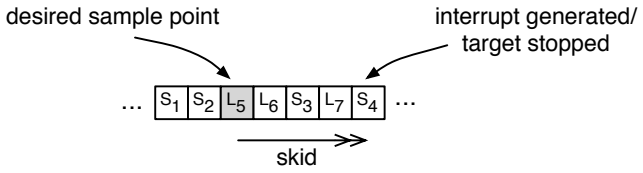


Figure 3.3: CPU skid. The load operations hardware counter is configured to overflow and stop the target on memory reference L_5. The overflow interrupt is deferred and delivered after some extra instructions are executed. The target is stopped on S_4.

After the reuse distance sample is taken, the monitor resumes the target execution, without delivering the *SIGIO* signal.

3.3.1 Processor Skid

In practice, there are problems with the hardware performance monitoring counters. There is a variable delay, in terms of instructions, between the actual overflow of a counter and the delivery of the overflow interrupt. This phenomenon is referred to as *skid* and means that the processor will execute for several more cycles after the actual counter overflow. Consequently, several instructions might be executed in the target before it is stopped. Figure 3.3 shows the skid effect. Note that this example only includes memory instructions. However the skid concerns all instructions and the target must not necessarily (most likely not) stop on a memory instruction.

In order to get an accurate profile of the target's data locality, the sampling mechanism must sample all memory references with the same probability. Nevertheless, the skid is likely to introduce a bias towards sampling certain types of memory instruction, for instance mainly instructions having a long latency.

To better understand the problem with CPU skid, lets look at an example. Consider a tight loop in which we are stepping through an array with 32 byte stride. Assume a cache line size of 64 bytes. The cache behavior when executing this loop is illustrated in Figure 3.4. Every second array access touches a new cache line, therefore causing a regular pattern of alternating cache misses and cache hits. The same scenario illustrated on a cycles time line is shown in Figure 3.5(a). Cache misses and cache hits result in long and short latency instructions, respectively.

Now lets consider the reuse distance for the array accesses. Considering the first load instruction, we see that it references the same cache line as the following instruction, thus having a reuse distance of zero. If we instead consider the second instruction, there are no following instructions that will reference the same cache line. The cache line will not be reused in the nearest future and we will have a long reuse distance. Again, we have a regular pattern of alternating short and long reuse distances. Figure 3.5(b) explains this.

If we also bring the skid effect into the picture, a problem emerges. Assume we want to sample the first array access, i.e. this access overflows the memory loads hardware counter. As illustrated in Figure 3.5(c), the CPU skid will defer the overflow interrupt for some cycles, causing us to sample on the next instruction instead. The result is that instead of sampling the

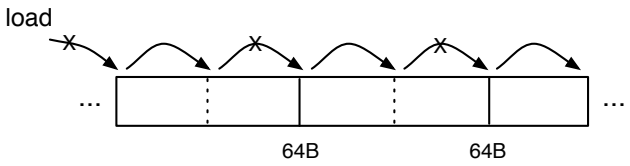


Figure 3.4: Consecutive array accesses in a tight loop. Assume 64 B cache lines. The arcs represent load operations, accessing the array with a 32 B stride. The loads with tick marks miss in the cache. Every second load will access a new cache line. The first load in the figure will miss in the cache (i.e. compulsory miss). Array data will be loaded into the corresponding cache line, and the second load will then hit in the cache. The third load accesses a new cache line and will again miss in the cache while the fourth load hit in the cache. The pattern of alternating cache misses and hits is repeated during the whole array access.

short reuse distance, the long reuse distance is sampled. If we want to the sample the second (long reuse) or third (short reuse) array access, the skid will in the same way cause us to miss the correct sample point. In both these cases we will end up sampling a long reuse, as seen in Figure 3.5(d). The overall effect of the skid in this loop example is that we consistently fail to sample the memory instructions having short reuses. Instead only the instructions with long reuses are sampled.

Note that the discussed scenario is just an example. The skid depends on the application and the effect can be vary. Nevertheless, the skid effect can severely distort the modeled cache behavior because the captured fingerprint gets an inaccurate distribution of reuse distances.

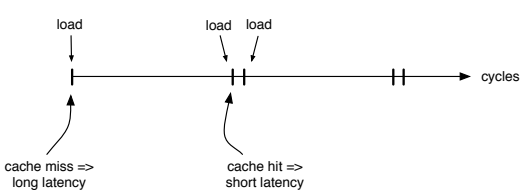
3.3.2 Resolving the Skid Problem

The skid effect must be considered and removed in order to get accurate results. The size of the skid is not constant, but depends on the instructions that are being executed. Measurements on the SPEC CPU2006 benchmarks [8] indicate that the skid is around 5-10 memory instructions on the Intel Nehalem architecture. This means that the processor, on average, continues to execute for a number of cycles, covering 5-10 memory instructions. We will discuss two possible solutions to the skid problem next.

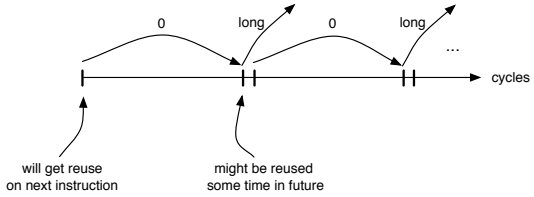
3.3.2.1 Skid Compensation

As previously described, the implementation has two sampling mechanisms running independently to sample both load and store operations. For simplicity, lets only consider the sampling of load operations. Also lets assume a constant sample period, *SamplePeriod*, i.e. the number of memory load instructions to execute before taking a sample. Let the maximum skid (in terms of memory instructions) be *MaxSkid*.

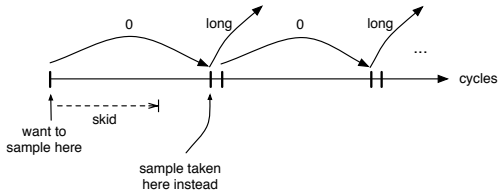
When a counter interrupt is generated, the target is stopped and we should take a sample. Normally, the sample point, *Sp*, would be:



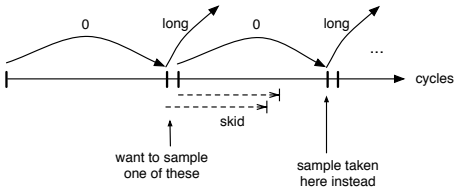
(a) Alternating array accesses will miss and hit in the cache, resulting in long and short latency, respectively.



(b) Alternating short and long reuse distances.



(c) Becasue of the skid, the long reuse is sampled instead of the short resuse.



(d) Similarily, a long reuse is sampled instead of a short reuse.

Figure 3.5: Example of the skid effect. An array is iterated with a 32 B stride on a machine with 64 B cache lines.

$$Sp = Time + SamplePeriod \tag{3.1}$$

where *Time* is the current value of the load counter. However, the skid allows the processor to execute a number of extra instructions before the counter interrupt is delivered. A straightforward solution to the problem would be to compensate for the skid by moving back the desired sample point.

The idea is to get the counter interrupt and stop the target execution *before* the processor has passed the desired sample point due to the skid. With skid compensation, the adjusted sample point, Sp', would instead be:

$$Sp' = Sp - MaxSkid \tag{3.2}$$

In 3.2, the sample point has been moved back by subtracting the maximum possible skid from the desired sample point.

When the load counter overflows, the interrupt is generated after some skid, but always before the desired sample point. The monitor process will intercept the *SIGIO* signal generated by perf_event, which means that a sample should be taken. To reach the desired sample point, the monitor must single-step the target process instruction by instruction.

Because we are using the memory load and store counters, counter values and the variables in Equation 3.2 are in terms of *memory* instructions. Therefore, the monitor must single-step over a certain number of memory instruction, before the desired sample point is reached. The required number of memory instructions to step over, $Steps$, is given by:

$$Steps = Time - Sp' \tag{3.3}$$

After single-stepping, the target stands at the desired sample point and a sample can eventually be taken.

A few issues were encountered when implementing the skid compensation solution. The ptrace API is used to single-step target instructions. The monitor process is notified when the target is stopped again after a single-step and the step is completed. This is indicated by a *SIGTRAP* signal from ptrace. However, during the single-stepping phase, the target process might be stopped for other reasons, e.g. other pending signals, requiring completely different actions. As an example, the cache line watchpoint mechanism discussed in next section, relies on *SIGSEGV* signals. The fact that the target process can be stopped by several different signals at the same time, is similar to a signal race condition. Therefore, the implementation must carefully manage pending signals to the target.

Two other issues are related to the perf_event API. First, perf_event can actually deliver the *SIGIO* sample start signal too early, i.e. before the number of executed memory instructions reaches the sample period. The reason behind this is not investigated, but might be due to some internal compensation mechanism in perf_event. There are no serious implications of this effect, other than that the monitor needs to single-step more instructions in these cases, which incurs some extra overhead. Second, there is another discovered anomaly in perf_event when updating the sample period for a counter. When the sample period is updated, the new period is not actually in effect immediately. Another sample is required before the new sample period is activated. This is of interest because it is desired to use a randomized sample period, which requires updating the sample period after every taken sample. This requires the monitor to keep track of the previous sample period.

The skid compensation method just discussed, solves the skid problem. Using this method, the monitor will always sample memory references with equal probability, thus capturing a very accurate data locality fingerprint. The downside of this method is the additional overhead incurred by the single-stepping. A single-step using ptrace involves multiple context switches which are expensive in terms of performance. Depending on how large the maximum skid is, every taken sample requires a bunch of single-steps which clearly degrades the performance. Nevertheless, for its great accuracy, this is the method chosen for the sampling mechanism in our implementation.

3.3.2.2 Double Trap

Instead of sampling a specific memory reference, one could sample the PC (program counter) of the instruction that references the memory. We refer to the method discussed here as the *double trap* method. This method makes use of the PEBS feature in recent Intel processors. As

described in Section 2.4, PEBS stores the register context when a hardware counter overflows. Therefore, the PC of the instruction causing the overflow is available despite the skid.

When a hardware counter overflows and the target process has stopped, after some skid, the approach is to insert a breakpoint at the PC where the overflow actually occurred. The breakpoint is inserted by replacing the first byte at the PC with a one-byte trap instruction (int3). The original byte is saved. Next, the target execution is resumed. Note that we are not trying to start a reuse distance sample immediately. The next time the target executes the same PC, the breakpoint will trigger. Since a real trap instruction is being executed, the target will stop immediately without skid, and the monitor process is informed via a *SIGTRAP* signal. The monitor will now remove the breakpoint by restoring the original byte. A reuse distance sample is then started and the target execution is resumed.

There are cases where the skid effect cannot be resolved by this method. Lets revise the skid example from earlier, looking at Figure 3.5(c) and Figure 3.5(d). The loop iterating through the array contains a single load instruction, thus all array references comes from the same PC. If we want to sample the first memory reference, the skid will defer the interrupt so the target is stopped just before the second memory reference. The PC of the desired sample point, i.e. where the counter actually overflowed, is read using PEBS and a breakpoint is inserted. But since this is the same PC as all the following memory instructions, the breakpoint will trigger immediately when the target execution is resumed. A reuse distance sample is then started. The effect is that a long reuse distance will be sampled instead of a short. Similarly, attempting to sample the second or third memory reference in this example will also result in sampling long reuse distances. Consequently, the same skid problem as pointed out previously still exists despite the double trap method.

Furthermore, there is a problem with PEBS known as *shadowing*. Shadowing is the effect of the small latency between a hardware counter overflow and the arming of the PEBS hardware. Shadowing is described in detail in [13]. The conclusion is that short latency instructions might be missed. Shadowing might seriously cripple the double trap method in a few scenarios.

As discussed, for some cases the double trap method is flawed. A common denominator for these cases is that they involve sequences of alternating short and long latency instructions, e.g. cache hits and misses, distributed over a tiny set of PCs. The example in Figure 3.5 is a degenerated case which demonstrates a worst-case scenario. Among the SPEC CPU2006 benchmarks, *libquantum* is one of few benchmarks were the skid effect is clearly visible when the double trap method is used. Figure 3.6 shows the working-set graphs for *libquantum* and *gamess*. Graphs for the skid compensation method, the double trap method as well as a reference graph are plotted. We can see that the accuracy with the double trap method for *libquantum* is poor. It is also clear that the accuracy with the skid compensation method is very close to the reference graphs.

Sampling using the double trap method is very fast compared to the skid compensation method described above since there is no need to single-step the target process when starting reuse distance samples. We estimated the speed up to roughly 5-10% when using the double trap method instead of the skid compensation method. However, due to the accuracy problem in some cases, the skid compensation method will still be used as the primary method for this implementation, despite the larger overhead.

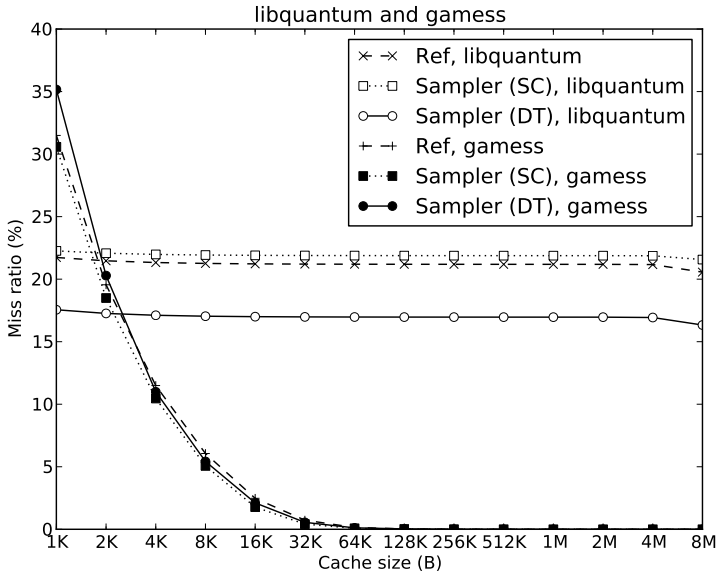


Figure 3.6: Working-set graphs for *libquantum* and *gamess*. Graphs for the skid compensation method and the double trap method are plotted. A reference graph is also plotted.

3.4 Watchpoints Mechanism

In order to detect cache line reuse, a mechanism for watching the cache memory is required. A common approach is to use code instrumentation to trace memory references. The downside of this approach is that it requires modification of the binary. We implement *cache line watchpoints* that monitor cache lines by using the memory management unit (MMU). This mechanism relies on paging and memory protection.

To setup a cache line watchpoint, the page that the cache line resides in is protected. Basically, this means that read and write permissions are removed while the watchpoint is active. To remove a cache line watchpoint, the original page permissions are restored for the page corresponding to the watched cache line.

When the target process reuses a watched cache line, the corresponding page is also referenced. Since read and write permissions are removed on this page, the operation is no longer permitted and the result is a segmentation fault. Segmentation faults generate *SIGSEGV* signals which are intercepted by the monitor process. When a cache line reuse is detected, the monitor terminates the reuse distance sample. Sample termination consists of the following basic steps:

1. **Check watchpoint filter**. Determine if the faulting memory address references a cache line that has a watchpoint on it.

2. **Remove cache line watchpoint**. Remove the matching cache line watchpoint and restore the original page permissions. When removed, re-execute the faulting instruction

by single-stepping it. If there are other active watchpoints in the same page, setup page protection again.

3. **Calculate reuse distance**. Read the current time, i.e. the number of memory references performed by the target so far. Calculate the reuse distance by using the time recorded previously at the sample start.

4. **Store sample**. Store sample in a sample file on disk or in memory.

After the reuse distance sample has been terminated, the monitor resumes the target execution, without delivering the *SIGSEGV* signal.

There is an obvious problem with this implementation of cache line watchpoints. To watch a cache line, a whole page is protected, although only a fraction of it is of interest. Consequently, all memory references to a page outside a watched cache line will result in segmentation faults. On the target system, the cache line size is 64 bytes and the page size is 4096 bytes. Typically, there are only one or a few active cache line watchpoints on the same page simultaneously. Thus, in most cases more than 95% of the data in a page containing a watchpoint is unnecessarily protected, resulting in many unwanted segmentation faults. These segmentation faults are referred to as *false positives*.

Every segmentation fault, including false positives, requires several context switches between the monitor and the target, e.g. when the monitor intercepts the *SIGSEGV* signal, removes page protections and re-executes the faulting instruction. Up to *eight* context switches may occur when the monitor handles a segmentation fault. Additionally, several system calls are required which results in a number of mode switches between kernel and user space. Altogether, these switches make the watchpoint mechanism fairly expensive. The previsouly discussed sampling mechanism with the skid compensation method has the similar problem. We will discuss more on the effects of large amounts of false positives and possible optimizations in Section 4 and 5.

3.5 Counting Memory References

With mechanisms to start and terminate reuse distance samples in place, the only remaining mechanism to measure reuse distances is a mechanism for counting memory references. We need to count the number of memory references between two consecutive references to the same cache line. To do this, the same hardware counters used by the sampling mechanism is utilized, that is the load and store operations counters. When a reuse distance sample is started and terminated, the current counter values are read and recorded. Let *BeginTime* and *EndTime* denote these values. The reuse distance, *ReuseDistance*, can then be calculated as the difference between the end time and the begin time:

$$ReuseDistance = EndTime - BeginTime - 1 \tag{3.4}$$

One is subtracted from the difference because the first nor the last memory reference are included in the reuse distance, see Definition 1 in Section 2.3.1.

There are some practical considerations when using the load and store hardware counters. According to the Intel System Developer's Manual [10], the counters are suppose to count instructions containing load and store operations. This is a vague description. Observations indicate that the counters also include indirect extra load and store operations caused by the executed instruction, i.e. page misses. The counters are also being incremented for more non-obvious events. Some of the observed anomalies are:

- Page faults increase the load and store counters.
- Trap instructions increase the load and store counters.
- System calls increase the load counter.
- Single-stepping will increase counters, even when stepping over non-memory instructions, e.g. ALU instructions only touching CPU registers.

If these "miscounts" are not considered, the measured reuse distances will be too large and thus inaccurate. To overcome this problem, we also count page faults, system calls, single-steps etc. to compensate for the extra counts when the sample begin and end times are acquired. This counter compensation method works well. Experiments showed that for short reuse distances, the number of memory references measured by the counters fully coincides with the true number of memory references performed by the target process. For very long reuse distances (in the order of 2^{25}), the numbers differed slightly. The error was usually less than 0.2%. Note that these experiments were made on micro benchmarks and not real-world applications.

3.6 Base Implementation

This section describes the base implementation of our cache analysis tool. All required mechanisms have been described in detail above. The sampling mechanism is implemented as described in 3.3 with the skid compensation method in Section 3.3.2.1. The watchpoint mechanism and memory reference counting are implemented as described in Section 3.4 and 3.5, respectively.

There are a few technical implementation details in the base implementation that have not been discussed yet. As described in Section 3.2, the ptrace API is used by the monitor to watch and control the target. All communication between monitor and target discussed so far, has been performed via the kernel and ptrace. A limitation in the Linux memory model complicates the cache line watchpoint mechanism. This mechanism is required to modify the access permissions of pages in the target process address space. Page permissions are modified with the *mprotect* system call in Linux. However, *mprotect* is restricted to work only with pages belonging to the calling process. This limitation, and security feature, forces us to call *mprotect* from the target process. In order to do so, we preload a shared library into the target process. The shared library simply spawns a thread for doing inter-process communication (IPC) with the monitor. When a reuse distance sample is started or terminated, the monitor issues commands over the link, telling the target to call *mprotect* to modify page permissions. The IPC link itself is implemented using local Unix domain sockets.

Using the MMU to implement the watchpoint mechanism, forces us to monitor certain system calls made by the target process. Inserting and removing cache line watchpoints involves setting read and write page permission flags. In addition to read and write permissions, there is also the execute permission. When a watchpoint is active, the read and write permission flags must be zeroed. When inactive, these flags must be restored. Otherwise proper execution of the target cannot be ensured. Consequently, the monitor must do some book-keeping of original page permissions for pages with active watchpoints. However, there is no obvious way of retrieving the current permission flags for a page in Linux, e.g. not via a system call. In our implementation, we use the maps file (*/proc/pid/maps*) in the proc file system [17] to read out the current virtual memory regions of the target, including the page permissions. As described in Section 3.4, every segmentation fault means that we need to alter the page permissions (regardless of whether it is a true or false positive). Since it is relatively expensive to parse the maps file for every segmentation fault, we only parse this file on start up and then keep a shadow copy of it during the target execution. System calls like *mprotect*, *mmap*, *munmap* can modify the virtual memory regions of a process. Therefore we need to monitor these system calls and investigate their function arguments to keep the shadow copy up to date. Dynamic library code might also be loaded into the target after some time in the execution. This requires us to do a fresh readout of the maps file.

4 Evaluation

The objective for this thesis work was to implement a cache modeling and analysis tool that is both accurate and very efficient. Therefore, we evaluate our implementation for accuracy and performance.

4.1 Methodology

Thirteen benchmarks from the SPEC CPU2006 benchmark suite were used for this evaluation. See Table 4.1 for details. The chosen set of benchmarks include both short- and long-running applications, applications stressing both the processor and the memory subsystem and applications with interesting phase behavior and working-set graphs.

Benchmark	**Input**
perlbench	diffmail.pl 4 800 10 17 19 300
bzip2	input.source 280
gcc	scilab.i
bwaves	
gamess	cytosine.2.config
milc	su3imp.in
zeusmp	
leslie3d	leslie3d.in
libquantum	1397 8
h264ref	-d foreman_ref_encoder_baseline.cfg
lbm	3000 reference.dat 0 0 100_100_130_ldc.of
astar	rivers.cfg
sphinx3	ctlfile . args.an4
povray	SPEC-benchmark-ref.ini
hmmer	nph3.hmm swiss41
omnetpp	omnetpp.ini

Table 4.1: Benchmarks and corresponding inputs used in the evaluation.

4.2 Experimental Setup

Table 4.2 describes the system that was used for evaluation.

Software	
Kernel	Linux 2.6.36
GCC	4.4.3
Hardware	
System	HP Z600 Workstation
Memory	6 GB ECC
Processor	Intel Xeon E5620@2.40GHz
Architecture	x86_64
Threads per core	2
Cores per socket	4
CPU sockets	1
NUMA nodes	1
CPU MHz	2395
L1d cache	32K
L1i cache	32K
L2 cache	256K
L3 cache	12288K

Table 4.2: Experimental setup

4.3 Accuracy

The cache modeling technique used by our cache analysis tool, StatCache, has been thoroughly evaluated [1] and shown to produce good results with satisfying accuracy. Therefore this modeling technique will not be evaluated again.

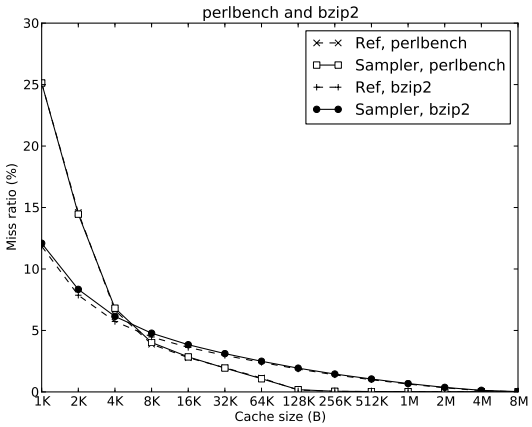
In this section we will evaluate the accuracy of our *implementation*. It will be mainly focused on the online component of the cache analysis tool, see Figure 3.1. Our implementation will be referred to as the *sampler*. The results from our sampler will be compared to a reference implementation, referred to as the *reference sampler* or simply the *reference*. The reference sampler implements the same cache modeling technique, but is based on *Pin* [14].

To evaluate the accuracy, we compare the estimated cache miss ratios from our sampler and the reference sampler. Differences are contrasted by plotting working-set graphs. Figure 4.1 and Figure 4.2 shows the working-set graphs for the benchmarks listed in Table 4.1. For every benchmark, a reference graph is plotted with a dashed line. The graphs are generated from the average of five evaluation runs.

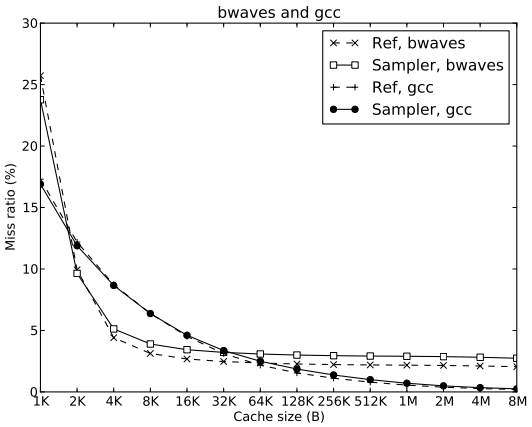
All benchmarks were sampled from the beginning to the end. We collected more than 100k samples for each benchmark with the reference sampler to get accurate and correct reference

data. We collected approximately 20k samples with our sampler. Because of the different execution times and code variation between the benchmarks, each benchmark was sampled with individual sample rates to collect the desired number of samples. Our tests have shown that 20k samples is sufficient to capture a representative fingerprint of application data locality. This is confirmed by the high accuracy demonstrated by Figure 4.1 and Figure 4.2. The error is on average well below one percentage point. In a few benchmarks, the error is slightly larger (1-1.5 percentage points), for instance *lbm* in Figure 4.2(a) and *zeusmp* in 4.1(c).

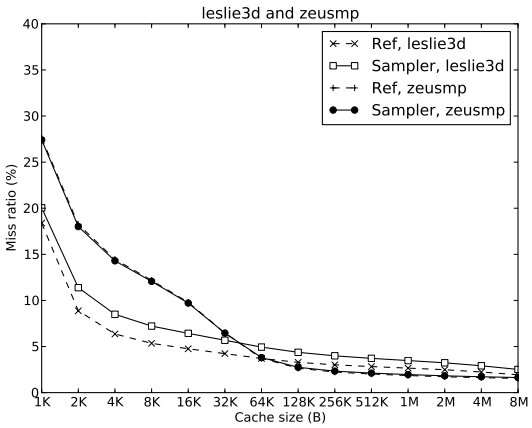
The minor differences between the reference sampler and our sampler have several possible causes. The skid problem discussed in Section 3.3.1 is a potential error source. The implemented skid compensation method *should* completely remove the skid effect. But the estimated maximum skid, used to move back the sampling points, might be incorrectly and/or optimistically set. An unnecessarily large estimation will add some extra overhead, while a too small estimation will not completely remove the skid and thus degrade the accuracy. Another possible error source is the problem with unreliable hardware counters, as discussed in Section 3.5. We have implemented a method for compensating for the extra counts that were observed. However, this method is not perfect. It does only compensate for the largest miscounts, caused by page faults, single-steps and system calls. It is likely that other events may affect the counters that we have not foreseen. Both the skid problem and the unreliable hardware counters can affect the shape of the distance histogram and therefore the accuracy.



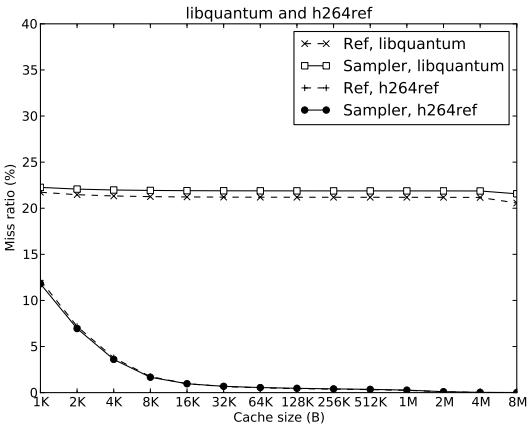
(a) *perlbench* and *bzip2*.



(b) *bwaves* and *gcc*.

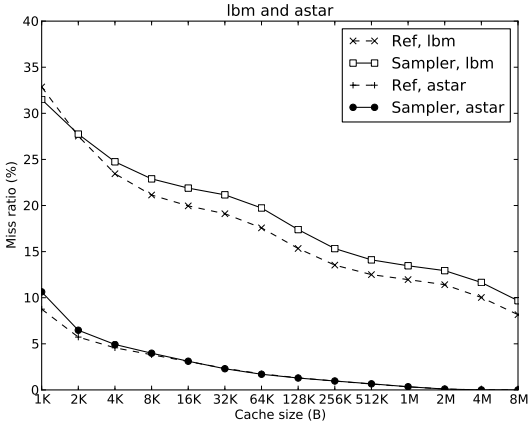


(c) *leslie3d* and *zeusmp*.

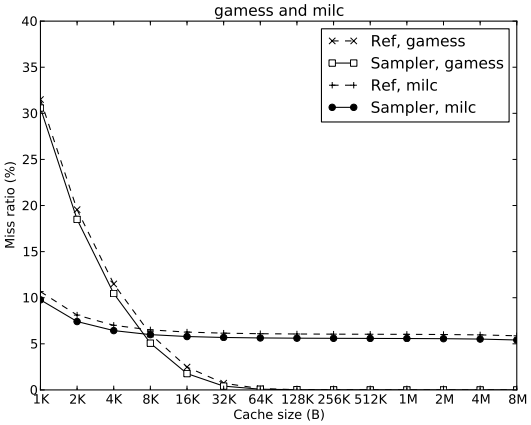


(d) *libquantum* and *h264ref*.

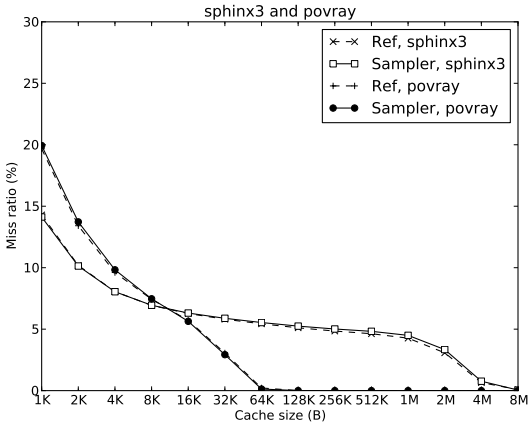
Figure 4.1: Working-set graphs generated from data locality information, captured by our sampler and a reference sampler. The reference data is based on over 100 k samples. The sampler data is based on around 20 k samples.



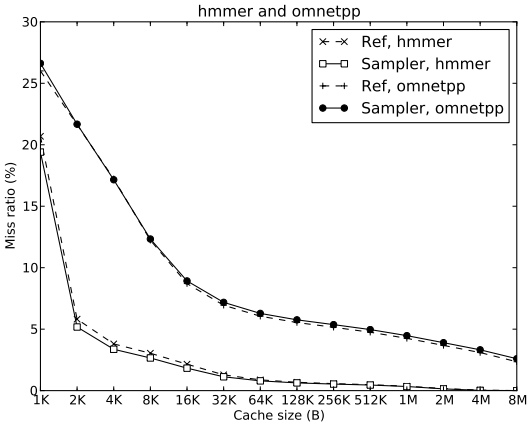
(a) *lbm* and *astar*.



(b) *gamess* and *milc*.

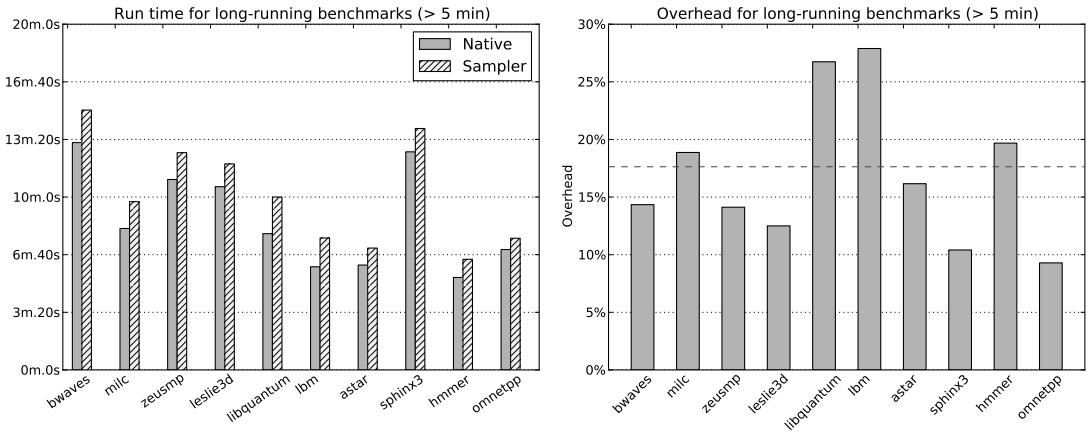


(c) *sphinx3* and *povray*.



(d) *hmmer* and *omnetpp*.

Figure 4.2: Working-set graphs generated from data locality information, captured by our sampler and a reference sampler. The reference data is based on over 100 k samples. The sampler data is based on around 20 k samples.



(a) Execution times. The native execution time and the execution time when sampling are shown for each benchmark.

(b) Overhead numbers for our sampler. The dashed horizontal line marks the average overhead which is approximately 17%.

Figure 4.3: Execution times and overhead for the long-running benchmarks when collecting around 20 k samples.

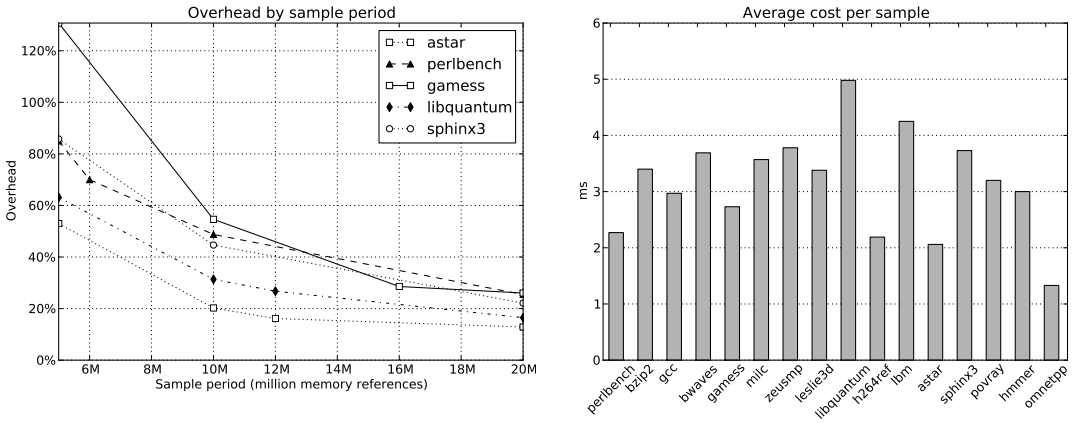
4.4 Performance

In this section we evaluate the performance of our cache analysis tool. We first present the run-time overhead for our tool. This is followed by a breakdown of the overhead where we identify the primary reasons for the overhead.

4.4.1 Run-Time Overhead

The benchmarks listed in Table 4.1 were run one time natively and one time with our tool, and the execution times were recorded. Around 20 k samples were collected. Some of the benchmarks have a rather short execution time, i.e. less than five minutes. In order to collect 20 k samples for these benchmarks before the execution finishes, we were forced to use a very aggressive sample rate, which in turn created a large run-time overhead. However, because of the short execution time for these benchmarks, it is acceptable to have a large overhead. Short-running applications will have a large overhead because of the forcibly high sample rate and not because of the implementation itself. Therefore their overhead numbers are not representative in the current evaluation where we wanted to collect a fixed number of samples.

Figure 4.3 shows execution times and overhead for all benchmarks with an execution time over five minutes. Native execution times are compared to execution times with sampling in Figure 4.3(a). The corresponding overheads are plotted in Figure 4.3(b). *omnetpp* has the smallest overhead just below 10% and *lbm* has the largest overhead around 27%. The average run-time



(a) Overhead as function of sample period.

(b) Average cost per sample in milliseconds.

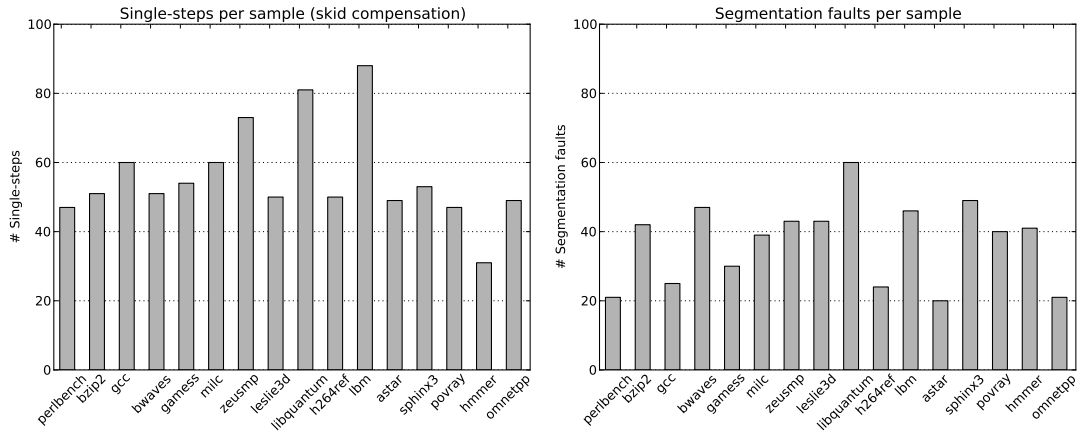
Figure 4.4: Run-time overhead as a function of sample period and average cost per sample.

overhead for our cache analysis tool was around 17%.

4.4.2 Overhead Breakdown

We will now investigate the reason behind the run-time overhead of our cache analysis tool. As already touched upon, the overhead is connected to the sample rate and the number of samples collected. The number of samples is, not surprisingly, linearly correlated to the sample rate. Doubling the sample rate, doubles the number of collected samples. A high sample rate means more interference with the native execution and will result in a large overhead. Figure 4.4(a) shows the overhead as a function of sample period for a few benchmarks. We see that the overhead roughly exponentially decreases with a longer sample period (lower sample rate). In order to do a fair comparison between the benchmarks, we have used the same constant sample rate for all benchmarks in this evaluation. Data presented in this section was generated using a sample period of 10^7, i.e. ten million memory references on average between samples. To understand the overhead and the variation between benchmarks, lets first look at the average cost for taking a sample (including both sample start and termination). Figure 4.4(b) shows the average cost per sample in milliseconds.

The overhead can be traced back to the sampling mechanism and watchpoint mechanism, described in Section 3.3 and 3.4. These mechanisms are the major sources of overhead and will be the focus for the rest of the discussion. Any other potential overhead in our cache analysis tool is regarded negligible. As discussed, the sampling mechanism with the skid compensation method, involves many single-steps before a reuse distance sample can be started. Every single-step requires context switches between the monitor and the target process, as well as switches between kernel and user space. These switches are expensive operations that add to the overhead.



(a) Average number of single-steps required to start a sample (the skid compensation method).

(b) Average number of segmentation faults per sample.

Figure 4.5: Single-steps and segmentation faults per sample.

Figure 4.5(a) shows the average number of single-steps taken per sample, when compensating for the skid on sample start. A high number of single-steps per sample means that we need to step the execution for many instructions before the desired sample point is reached. In other words, the actual skid was short. Similarly, a low number means a long skid and we only need to step a few instructions before reaching the desired sample point. We see in the figure that *lbm* requires many single-steps per sample and therefore has a short skid. *hmmer* requires the least number of single-steps which implies a long skid.

The watchpoint mechanism also adds to the overhead. Whole pages are protected although only tiny parts of the pages are of interest. Every memory reference to a page containing a monitored cache line will result in a segmentation fault. This occurs regardless of whether the actual cache line or other memory in the page was referenced. The segmentation faults are expensive and also requires multiple context switches. A more detailed discussion is given in Section 3.4. Figure 4.5(b) shows the average number of segmentation faults per sample. Ideally, there would only be one segmentation fault per sample, i.e. when cache line reuse is detected. However, due to false positives, we have several segmentation faults per sample. We see that *libquantum* has most segmentation faults per sample. This means that *libquantum* more frequently references memory outside cache lines on protected pages, which also suggests that *libquantum* has a relatively poor data locality.

Benchmarks with large amounts of single-steps or segmentation faults per sample will have a large run-time overhead. This can be seen by comparing Figure 4.3(b) with Figures 4.5(a) and 4.5(b). For instance, study *lbm* and *libquantum*. In order to more clearly see how the number of single-steps and segmentation faults affect the cost per sample, we can calculate the cost per

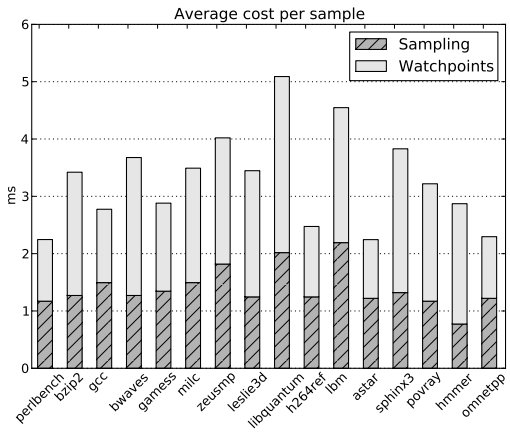


Figure 4.6: Average cost per sample in milliseconds. The average costs of the sampling mechanism and the watchpoint mechanism are shown. The costs are calculated from Equation 4.1 and data from Figure 4.5.

single-step and the cost per segmentation fault. The average cost per sample for a benchmark can be expressed as:

$$C(n_s, n_f) = \alpha n_s + \beta n_f \tag{4.1}$$

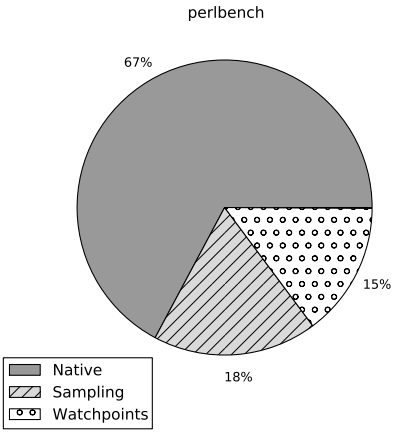
where n_s is the average number of single-steps per sample, n_f is the average number of segmentation faults per sample, α is the cost of taking a single-step and β is the cost of a segmentation fault. Note that we only consider single-steps that are related to the sampling mechanism. By solving Equation 4.1 using data from evaluation runs, we can deduce estimations of how expensive the sampling and watchpoint mechanisms are. It is reasonable to believe that the cost parameters α and β are roughly the same for all benchmarks. Intuitively, the cost of a single-step or a segmentation fault should be an application independent system property.

Equation 4.1 was solved for all benchmarks and the cost parameters α and β are in fact nearly constant. The cost for a single-step (sampling mechanism) and the cost for a segmentation fault (watchpoint mechanism) was calculated to 0.025 ms and 0.051 ms, respectively. Figure 4.6 again shows the average cost per sample, but with the cost of each mechanism included.

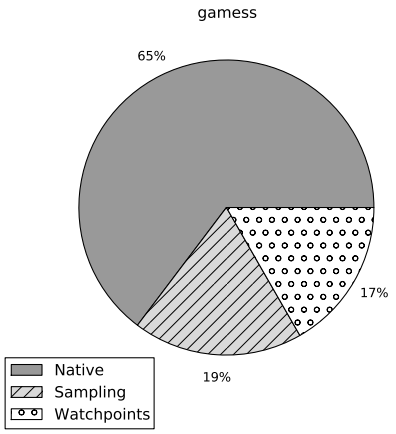
The overall impact on performance by the sampling and watchpoint mechanisms depends on the executed benchmark. The overhead variations between the benchmarks are explained by the number of single-steps and segmentation faults. As we saw in Figure 4.5(a) and 4.5(b), these numbers can differ significantly between different benchmarks. In some benchmarks, the overhead is dominated by the sampling mechanism while the overhead in other benchmarks are primarily explained by the watchpoint mechanism. We can breakdown the overhead and quantify the impact of the two mechanisms by comparing native execution time, execution time with only the sampling mechanism enabled and execution time with both sampling and watchpoint mechanisms enabled. Figure 4.7 shows this division for a few benchmarks. We can conclude

that the watchpoint mechanism is slightly more expensive in general. We can also see that the cost of single-steps and segmentation faults, as shown in Figure 4.6, are directly reflected in the overhead breakdown in Figure 4.7.

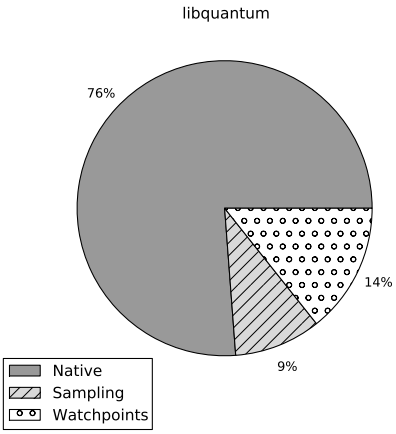
We have seen that the sampling and the watchpoint mechanisms are the two main reasons behind the run-time overhead of our cache analysis tool. The next chapter will discuss some optimizations that can potentially minimize this overhead.



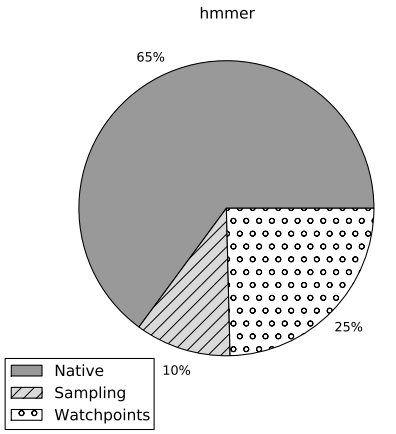
(a) Overhead distribution of perlbench.



(b) Overhead distribution of gamess.



(c) Overhead distribution of libquantum.



(d) Overhead distribution of hmmer.

Figure 4.7: Breakdown of the run-time overhead for some benchmarks. The dark segment in the pie charts represents the native execution time. The striped and dotted segments are the overhead for the samlping mechanism and the watchpoint mechanism, respectively.

5 Future Work

There are some future work that can improve our cache analysis tool. This section discusses a number of optimizations that can reduce the run-time overhead. As seen in the evaluation in Section 4.4.2, the overhead is largely explained by the mechanism for starting and terminating reuse distance samples. More specifically, the skid compensation method and the cache line watchpoints are the major performance hogs.

5.1 Optimized Sampling

The overhead when starting new reuse distance samples comes from the need to single-step the target process many times in order to overcome the processor skid problem. This skid compensation method, described in Section 3.3.2.1, involves many expensive context switches between the monitor and target process, as well as mode switches between kernel and user space. The double trap method described in Section 3.3.2.2 was an attempt to optimize the sample starts by removing the need for single-stepping. The idea was to use breakpoints and the PEBS feature instead. However, accuracy problems prohibited its use.

Another possible optimization is to keep the skid compensation method but move the whole sampling mechanism into kernel space. This would remove most of the expensive switches between kernel and user space. An implementation would preferably be in the form of a kernel module that handles all the single-stepping without bothering user space.

5.2 Optimized Cache Line Watchpoints

The watchpoint mechanism is the other major source of overhead. The reason for this overhead is similar to that in the skid compensation method; many expensive switches between the monitor and target process and between kernel and user space. As discussed in Section 3.4, the problem is the larger number of *false positives*. All references to memory on pages that have cache line watchpoints on them, will generate segmentation faults.

The cache line watchpoint handling can also be optimized by moving it into kernel space. The watchpoints will still be built using the MMU and page protection. The main idea is to have a kernel module that intercepts segmentation fault signals addressed to the target process and filters out all false positives. The monitor process will then only see a segmentation fault being

delivered to the target when a cache line is reused for real. The kernel module could use the proc file system as an interface to register cache line watchpoints. This kind of watchpoint could also be useful for other applications, e.g. debuggers. In-kernel watchpoints are supported by some operating systems, e.g. Solaris, and are used by the StatCache implementation in [2].

A positive side-effect of moving the cache line watchpoint handling into kernel space is that the IPC communication discussed in Section 3.6 will be superfluous. When in kernel space, page permissions can be set for any process without limitations. Consequently, it is unnecessary to do this inside the target address space and preloading an IPC thread will therefore no longer be needed. Furthermore, the monitor no longer needs to keep a shadow copy of the target's memory maps file. Also, it does not need to listen for system calls that might modify this file.

In-kernel cache line watchpoints is an optimization with great potential that should considerably reduce the overhead associated with sample termination.

Ideally, we would like true hardware watchpoints on the cache lines. Some processors support hardware memory watchpoints, but these are limited in numbers and in the range of memory they can monitor. Implementing watchpoints using page protection will always have some overhead due to the false positives. This remains true even if the watchpoint handling is moved into the kernel. An implementation that more resembles true hardware cache line watchpoints might be possible by exploiting the error-correction code (ECC) bits in the RAM modules. The idea is to simulate an ECC error on the desired memory range when setting up a watchpoint. The consequence will be that the hardware will signal the operating system about the error as soon as the memory range is referenced again. This event should then be caught by the watchpoint mechanism in order to detect reuse. This is an advanced optimization that might be cumbersome to implement. However, it is an optimization that probably would reduce the watchpoint mechanism overhead substantially.

5.3 Phase-Guided Sampling

The performance of our cache analysis tool is mainly explained by the run-time overhead of the sampling mechanism and the cache line watchpoints. The performance can of course be enhanced by optimizing these mechanisms and some optimizations have already been suggested. By exploiting phase behavior inherent in many applications, sampling re-occurring and redundant periods of execution can be avoided. This was discussed briefly in Section 2.5.

Sampling windows were briefly discussed in Section 2.3.3. Basically, we assume that the cache miss ratio is constant in the probability theory of the cache model. This is generally not true and the execution is split into sampling windows that are sufficiently short to assume a constant miss ratio. A benefit with phase-guided sampling is that the phases will be natural sampling windows.

One problem with sampling is knowing how many samples to take. Taking many samples requires a high sample rate. This would give us a very accurate application data locality fingerprint but, unfortunately, it would also have a negative impact on performance. On the other hand, taking few samples, requiring a low sample rate, offers good performance but might result in poor accuracy. Our tests indicate that about 20 k samples are more than sufficient to get good accuracy. The number of samples is not critical per se, what really matters is that the samples are

representative for the application and capture all parts of the execution. Sampling using phase detection algorithms can assist in the decision of when to sample. It is plausible that the overall run-time overhead can be reduced for some applications with phase-guided sampling.

The main objective of our cache analysis tool in this thesis has been to determine the *average* cache miss ratio over a whole program execution. While phase-guided sampling can potentially reduce the overhead in this case, another more direct approach is to simply adjust the sample rate so only a sufficient number of samples is collected. 20 k samples were suggested above as *more than sufficient*. In fact, if we are only after the average cache miss ratio, a number as low as 1 k samples still gives us high accuracy along with a major reduction in overhead. The average for the benchmarks evaluated in Chapter 4 is then $< 1\%$.

The average miss ratio might be very misleading if only a specific time slice of the execution is considered. Phase-guided sampling is most valuable if we want to analyze the cache behavior over time. In this case, the cost of sampling is constant, i.e. we need to collect a fixed number of samples or have a fixed sample rate to capture enough information about a phase. Determining miss ratio per phase would be very expensive compared to the average miss ratio since we need many more samples in each phase. However, using phase detection algorithms to guide the sampling, we only need to collect samples once per phase and can ignore all following redundant phases. Also, phase-guided sampling should be good for correlating cache behavior to source code.

An experimental implementation of phase-guided sampling was built on top of the base implementation. The prototype has not yet been fully evaluated. Nevertheless, the preliminary results were promising and the prototype allows us to efficiently analyse cache behavior over time.

6 Summary and Conclusions

In this thesis, we have studied and implemented a method for efficiently analysing application cache behavior. The method uses a probabilistic cache model named StatCache. This model uses a sparse fingerprint of application data locality as input. The input is based on reuse distances between cache lines. The model uses probability theory and numerical methods to transform the input into cache miss ratios. By varying architecture-specific parameters, e.g. cache size, miss ratios for a range of different architectures can be calculated. Working-set graphs can then be plotted from these data.

The implementation of our cache analysis tool targets a modern Intel x86-64 architecture and the Linux operating system. The implementation has primarily focused on the data acquisition component of the cache analysis method. We examine the memory reference stream of the studied application to capture the reuse distance information that make up the data locality fingerprint. This is accomplished by utilizing hardware counters and standard debug APIs. A sampling technique is used to capture a light-weight fingerprint. A watchpoint mechanism for detecting cache line reuse is implemented using memory page protection.

We have evaluated our implementation for accuracy and performance using applications from the SPEC CPU2006 suite. Working-set graphs from our implementation were compared with reference graphs. The accuracy of our implementation was very high. The difference in miss ratio compared to a slow reference implementation was usually below one percentage point. We compared native execution times with the execution times when running with our cache analysis tool. The performance is good; the average run-time overhead was around 17%.

A number of optimizations that could reduce the overhead further were proposed. The sampling mechanism and the watchpoint mechanism can be moved into the kernel to avoid expensive mode switches between kernel and user space. Phase-guided sampling is suggested as a key optimization where application phase behavior is used to determine when to sample memory references. We have built a test implementation of this optimization and the preliminary results were promising.

References

[1] E. Berg and E. Hagersten. Statcache: a probabilistic approach to efficient and accurate data locality analysis. In *Performance Analysis of Systems and Software, 2004 IEEE International Symposium on - ISPASS*, pages 20 – 27, 2004.

[2] E. Berg and E. Hagersten. Fast data-locality profiling of native execution. In *Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '05, pages 169–180, New York, NY, USA, 2005. ACM.

[3] S. Carr, K. S. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. In *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, ASPLOS-VI, pages 252–262, New York, NY, USA, 1994. ACM.

[4] T. Conte, M. Hirsch, and W.-M. Hwu. Combining trace sampling with single pass methods for efficient cache simulation. *Computers, IEEE Transactions on*, 47(6):714 –720, June 1998.

[5] L. DeRose, K. Ekanadham, J. Hollingsworth, and S. Sbaraglia. Sigma: A simulator infrastructure to guide memory analysis. In *Supercomputing, ACM/IEEE 2002 Conference*, page 1, November 2002.

[6] D. Eklov and E. Hagersten. Statstack: Efficient modeling of lru caches. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pages 55 –65, March 2010.

[7] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Trans. Program. Lang. Syst.*, 21:703–746, July 1999.

[8] J. L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34:1–17, September 2006.

[9] M. D. Hill and A. J. Smith. Evaluating associativity in cpu caches. *IEEE Trans. Comput.*, 38:1612–1630, December 1989.

[10] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2*.

[11] R. Kessler, M. Hill, and D. Wood. A comparison of trace-sampling techniques for multi-megabyte caches. *Computers, IEEE Transactions on*, 43(6):664 –675, June 1994.

[12] A. Lebeck and D. Wood. Cache profiling and the spec benchmarks: a case study. *Computer*, 27(10):15 –26, Oct. 1994.

[13] D. Levinthal. Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors. Technical report, Intel Corporation, 2008.

[14] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[15] P. S. Magnusson, F. Dahlgren, H. Grahn, M. Karlsson, F. Larsson, F. Lundholm, A. Moestedt, J. Nilsson, P. Stenström, and B. Werner. Simics/sun4m: a virtual workstation. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '98, pages 10–10, Berkeley, CA, USA, 1998. USENIX Association.

[16] J. Mellor-Crummey, R. Fowler, and D. Whalley. Tools for application-oriented performance tuning. In *Proceedings of the 15th international conference on Supercomputing*, ICS '01, pages 154–165, New York, NY, USA, 2001. ACM.

[17] E. Mouw. *Linux Kernel Procfs Guide*. Faculty of Information Technology and Systems, Delft University of Technology, The Netherlands.

[18] N. Nethercote. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, University of Cambridge, November 2004.

[19] E. Perelman, G. Hamerly, and B. Calder. Picking statistically valid and early simulation points. In *Parallel Architectures and Compilation Techniques, 2003. PACT 2003. Proceedings. 12th International Conference on*, pages 244 – 255, September 2003.

[20] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder. Using simpoint for accurate and efficient simulation. In *Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '03, pages 318–319, New York, NY, USA, 2003. ACM.

[21] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod. Using the simos machine simulator to study complex computer systems. *ACM Trans. Model. Comput. Simul.*, 7:78–103, January 1997.

[22] F. Sanchez, A. Gonzalez, and M. Velero. Static locality analysis for cache management. In *Parallel Architectures and Compilation Techniques., 1997. Proceedings., 1997 International Conference on*, pages 261 –271, November 1997.

[23] A. Sandberg, D. Eklöv, and E. Hagersten. Reducing Cache Pollution Through Detection and Elimination of Non-Temporal Memory Accesses. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[24] A. Sembrant. Low overhead online phase predictor and classifier. Master's thesis, Department of Information Technology, Uppsala University, Sweden, January 2010.

[25] D. A. Wood, M. D. Hill, and R. E. Kessler. A model for estimating trace-sample miss ratios. In *Proceedings of the 1991 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, SIGMETRICS '91, pages 79–89, New York, NY, USA, 1991. ACM.